VILNIUS GEDIMINAS TECHNICAL UNIVERSITY
FACULTY OF MECHANICS
DEPARTMENT OF MECHATRONICS AND ROBOTICS

Alejandro Perez Sanmartin

RESEARCH OF ROBOT TRAJECTORY CONTROL BY OPTICAL METHOD

Master‘s degree Thesis

Mechatronics Systems study programme, state code 621H73002
Production and Manufacturing Engineering study field

Vilnius, 2017

VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

FACULTY OF MECHANICS

DEPARTMENT OF MECHATRONICS AND ROBOTICS

APPROVED BY
Head of Department
_____
(Signature)
Vytautas Bučinskas
(Name, Surname)
25-05-2017
(Date)

Alejandro Perez Sanmartin

RESEARCH OF ROBOT TRAJECTORY CONTROL BY OPTICAL METHOD

Master's degree Thesis

Mechatronics Systems study programme, state code 621H73002

Production and Manufacturing Engineering study field

Supervisor Assoc Prof Dr Vytautas Bučinskas _____ 25-05-2017
(Title, Name, Surname) (Signature) (Date)

Consultant _____ _____ _____
(Title, Name, Surname) (Signature) (Date)

Consultant _____ _____ _____
(Title, Name, Surname) (Signature) (Date)

Vilnius, 2017

VILNIUS GEDIMINAS TECHNICAL UNIVERSITY
FACULTY OF MACHANICS
DEPARTMENT OF MECHATRONICS AND ROBOTICS

Study area – Production and Manufacturing Engineering

Study programme – Mechatronics systems, state code 621H73002

APPROVED BY:
Head of Department

(Signature)
Vytautas Bučinskas
(Name, Surname)

(Date)

TASK TO PREPARE MASTER'S FINAL WORK

11-04-2017 Nr. 04
Vilnius

For student Alejandro Perez Sanmartin

Final Work Title: Research of robot trajectory control by optical method.

Approved 10 of April 2017 by Dean's decree No 166me

The Work has to be completed by 19 of May 2017.

THE FINAL TASK:

1. Perform literature overview and analysis of methods and test rigs of the presented topic;
2. Create methodology for theoretical and experimental research;
3. Perform theoretical and experimental research;
4. Perform analysis of researches, present results conclusions and recommendations.

Academic Supervisor ..... prof. dr. Vytautas Bučinskas
(Signature) (Name, Surname)

The task accepted

.....
(Signature)
Alejandro Perez Sanmartin
(Name, Surname)
11-04-2017
(Date)

(the document of Declaration of Authorship in the Final Degree Paper)

VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

Alejandro Sanmartin Perez, 20170029

(Student's given name, family name, certificate number)

Faculty of Mechanics

(Faculty)

Mechatronics Systems, MSfmu-15

(Study programme, academic group no.)

DECLARATION OF AUTHORSHIP
IN THE FINAL DEGREE PAPER

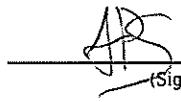
May 23, 2017

I declare that my Final Degree Paper entitled „Research of robot trajectory control by optical method" is entirely my own work. The title was confirmed on April 10, 2017 by Faculty Dean's order No. 166me. I have clearly signalled the presence of quoted or paraphrased material and referenced all sources.

I have acknowledged appropriately any assistance I have received by the following professionals/advisers: Assoc Prof Dr Vytautas Bučinskas.

The academic supervisor of my Final Degree Paper is Assoc Prof Dr Vytautas Bučinskas.

No contribution of any other person was obtained, nor did I buy my Final Degree Paper.



(Signature)

Alejandro Sanmartin Perez

(Given name, family name)

Vilnius Gediminas Technical University
Faculty of Mechanics
Department of Mechatronics and Robotics

ISBN ISSN
Copies No. .....
Date .....-.....-.....

Master Degree Studies **Mechatronics Systems** study programme Master Graduation Thesis 4

Title	**Research of robot trajectory control by optical method**
Author	**Alejandro Sanmartin Perez**
Academic supervisor	**Vytautas Bučinskas**

Thesis language: English

Annotation

This project presents the integration of Kinect 3D camera in Kuka Youbot and the capabilities that this provides to Kuka with a real example case. Specifically, the algorithm, which has been developed, recognizes a red object and provides the coordinates of the centroid to Kuka Youbot Arm. The method has been implemented using Point Cloud Library and color and depth segmentation techniques. Particularly, registered points are processed and transformed to HSV color space avoiding the effect of lightness in color segmentation. Furthermore, Kuka Youbot model in MoveIt is used to plan and simulate the movement of the robot from the home position to the centroid point of the object.

Keywords: Kuka, Youbot, Object Detection, Color Segmentation, Kinect, 3D camera, PCL, MoveIt!, Motion planning, Inverse Kinematics

INDEX

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

- KUKA: Keller und Knappich Augsburg
- RGB: color model meaning Red, Green and Blue.
- RGBD: color model meaning Red, Green and Blue and Depth information.
- 1D: One dimension.
- 2D: Two dimensions.
- 3D: Three dimensions.
- VGTU: Vilniaus Gedimino Technikos Universitetas.
- IR: Infrared
- CMOS: Complementary metal–oxide–semiconductor
- Nan: Not a number
- PCL: Point Cloud Library
- PCD: Point Cloud Data
- SLAM: Simultaneous Localization And Mapping
- XML: eXtensible Markup Language
- LISP: Locator/ID Separation Protocol
- Rviz: ROS visualizer
- URDF: Universal Robotic Description Format
- SRDF: Semantic Robotic Description Format
- TF: Transform Information
- OMPL: Open Motion Planning Library

INTRODUCTION

The thesis is part of a larger ongoing research project of the Department of Mechatronics and Robotics of VGTU. The goal of that project is to develop software for KUKA YouBot and contribute to the study of advanced robotics control. Specifically, in this thesis Kinect camera is used for detecting objects and planning the movement that Kuka YouBot will do.

Research Method

Firstly, it will be a short literature review of the types of sensors that are currently being used in robotics control. After that, characteristics of the chosen sensor, Kinect camera, and robot, Kuka YouBot, will be detailed. Then, there will be a theoretical research where the software that will be used to solve the problem will be explained. Moreover, a method will be developed and theoretical conclusions will be commented. Finally, experimental research will be done. Experiment will be detailed and results will be analyzed.

Objectives of the research

The main objective of the thesis is to incorporate the capabilities of a Kinect 3D camera to the Kuka YouBot installed in the Laboratory of Mechatronics of VGTU. It will be developed a method that will provide vision to Kuka YouBot and create a base for more complex future studies of robot trajectory control using Kinect and YouBot. Furthermore, in this research a real engineering problem will be solved, that is picking with YouBot a red object detected by Kinect.

Motivation

The personal motivation of this research is to complete my studies in Industrial Technologies researching in some specific field of Engineering, in this case mechatronics and robotics. My purpose accepting this challenge is to learn and to deal with a specific problem that an engineer has to solve. Moreover, with this research, I have the possibility to improve my skills in C/C++ programming an algorithm that will be implemented in a top technology field such as robotics. It is a good opportunity, which I could not let pass, for thinking of my future and my entrance in the world of employment.

Conference

The results of this master thesis research were presented in the conference: *Lithuanian Junior Researchers: "Science - Future of Lithuania"* celebrated in Vilnius in April 2017.

1 LITERATURE REVIEW

1.1 State of art

Nowadays, globalization and the increase of competence has forced companies to innovate and to improve automated processes to reduce direct costs and make them more competitive. Multinationals as Amazon or Toyota has invested big amounts of money in the development of new autonomous robots, specially picking up robots (Fig. 1.1.). As a result, Industry 4.0 has exponentially grown, computerizing the world factory and introducing robots into production and logistics. In addition, the development of new technologies, such as voice recognition, 3D cameras, etc., have made the integration of sensors into robots a very important field of research due to the complexity and the huge challenge that it supposes.

Fig. 1.1. Amazon picking challenge (http://www.sogotechnews.com)

To get a robot to perform its task with the right precision, speed and intelligence, it will require knowledge of its own state and the state of its environment. This information is obtained by sensors.

A sensor is a device that detects and responds to some type of input from the physical environment. The specific input could be light, heat, motion, moisture, pressure, or any one of a great number of other environmental phenomena. The output is generally a signal that is converted to human-readable display at the sensor location or transmitted electronically over a network for reading or further processing. (http://whatis.techtarget.com/definition/sensor)

In robots, the information related to its internal state (battery level, wheel position, joint angle, etc.) is achieved with the so-called **proprioceptive sensors**, while that relating to the state of its environment is acquired with the **exteroceptive sensors**. Furthermore, robot sensors can be classified in active and passive depending on them receiving or emitting energy. In table 1 take, general robot sensor classification depending on his use is done.

Table 1.1. Types of sensors and characteristics (*Sensors for Robots*, Czech Technical University)

General Classification	Typical Use	Sensor	Proprioceptive / Exteroceptive	Active / Passive
Tactile sensors	Detection of physical contact	Contact switches, bumpers	Exteroceptive	Passive
	or closeness: security switches	Optical barriers	Exteroceptive	Active
		Noncontact proximity sensors	Exteroceptive	Active
Wheel/motor sensors	Wheel/motor speed and position	Brush encoders	Proprioceptive	Passive
		Potentiometers	Proprioceptive	Passive
		Synchros, resolvers	Proprioceptive	Active
		Optical encoders	Proprioceptive	Active
		Magnetic encoders	Proprioceptive	Active
		Inductive encoders	Proprioceptive	Active
		Capacitive encoders	Proprioceptive	Active
Heading sensors	Orientation of the robot in relation	Compass	Exteroceptive	Passive
	to a fixed reference frame	Gyroscopes	Proprioceptive	Passive
		Inclinometers	Exteroceptive	Active / Passive
Ground-based beacons	Localization in a fixed reference frame	GPS	Exteroceptive	Active
		Active optical or RF beacons	Exteroceptive	Active
		Active ultrasonic beacons	Exteroceptive	Active
		Reflective beacons	Exteroceptive	Active
Active ranging	Reflectivity, tife-of-flight,	Reflectivity sensors	Exteroceptive	Active
	geometric triangulation	Ultrasonic sensor	Exteroceptive	Active
		Laser rangefinder	Exteroceptive	Active
		Optical triangulation (1D)	Exteroceptive	Active
		Structured light (2D)	Exteroceptive	Active
Motion/speed sensors	Speed relative to fixed or movin objects	Doppler radar	Exteroceptive	Active
		Doppler sound	Exteroceptive	Active
Vision-based sensors	Visual ranging, whole-image analysis,	CCD/CEMOS camera(s)	Exteroceptive	Passive
	segmentation, object recognition	Visual ranging packages	Exteroceptive	Passive
		Object tracking packages	Exteroceptive	Passive

In that project, the Microsoft Kinect camera is integrated in Kuka YouBot. Kinect incorporates vision-based sensors (CEMOS camera) and infrared sensors as it will be explained in detail in chapter 1.2.

1.2 Kinect

The Microsoft Kinect (Fig. 1.2.) is a consumer device originally designed by Microsoft as a peripheral for the Xbox game system and Windows PCs. Kinect's low price and the good quality of the depth information that is produced makes Kinect an attractive sensor for robotics research. As a result, a variety of open-source drivers have been developed for the Kinect. (*Mobile Manipulation for the KUKA youBot Platform,* Worcester Polytechnich Institute)

Fig. 1.2. Microsoft Kinect (*Kinect*, https://www.xataka.com)

It includes the following components and sensors (Fig 1.3.) (https://msdn.microsoft.com):

- An RGB camera that stores three channel data with a 1280x960 resolution. This makes possible to obtain RGB image.
- An infrared (IR) emitter and an IR depth sensor. The emitter emits infrared light beams and the depth sensor reads the IR beams reflected back to the sensor. This makes possible to obtain Depth information.
- A multi-array microphone, which contains four microphones for capturing sound.
- A 3-axis accelerometer configured for a 2G range, where G is the acceleration due to gravity. It is possible to use the accelerometer to determine the current orientation of the Kinect.

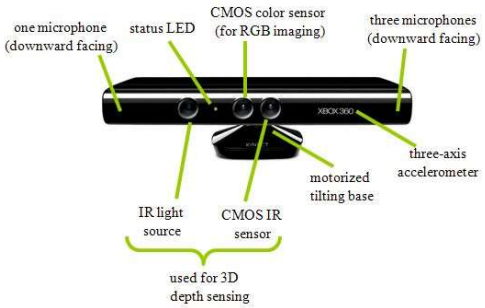


Fig. 1.3. Microsoft Kinect sensors (*Kinect Chapters 1 and 2*, http://fivedots.coe.psu.ac.th)

In table 2 take, the technical specifications of Kinect are detailed.

Table 1.2. Technical specifications of Microsoft Kinect (*Kinect for Windows Sensor Components and Specifications,* https://msdn.microsoft.com,)

Kinect	Array Specification
Viewing angle	Vertical: 43º Horizontal: 57º
Vertical tilt range	± 27º
Frame rate	30 Frames per second
Audio format	16-kHz, 24-bit mono pulse code modulation (PCM)
Audio input characteristics	A four-microphone array with 24-bit analog-to-digital converter (ADC) and Kinect-resident signal processing including acoustic echo cancellation and noise suppression
Accelerometer characteristics	A 2G/4G/8G accelerometer configured for the 2G range, with a 1° accuracy upper limit.

In Fig. 1.4. angles of vision of Kinect are shown:

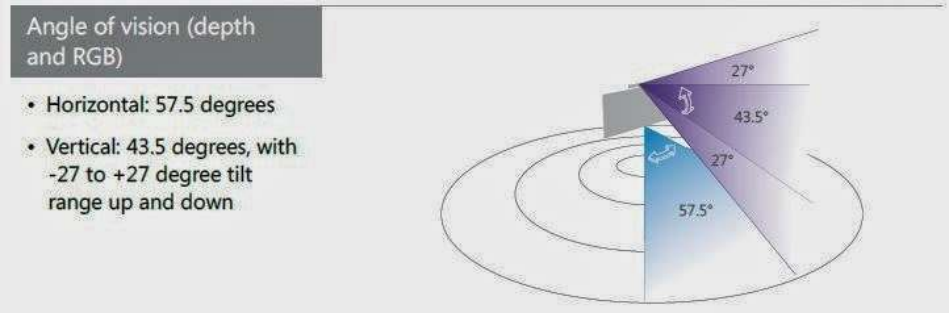


Fig. 1.4. Angles of vision of Microsoft Kinect (http://thethirdeye-idea.blogspot.lt)

The Kinect derives depth information from an environment by projecting a grid of infrared points in a predictable pattern using a diffuser and diffractive element of IR light. The IR Depth Sensor observes the scene and receives the signal.

The depth data is calculated with the triangulation of each speckle between the projected pattern and the observed pattern. The distortion of light pattern allows the Kinect to compute the 3D structure. The coding has to be unique per position in order to recognize each point in the pattern. (https://fabrizio89.wordpress.com/kinect-1-generation)

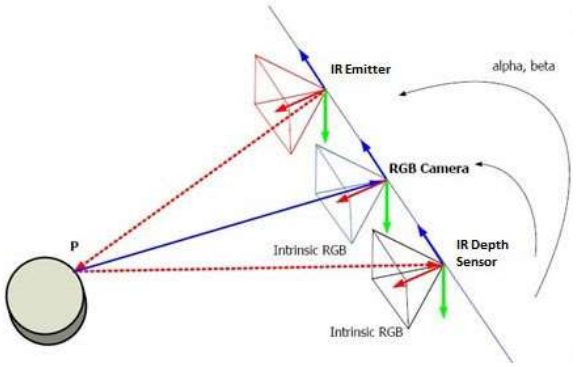


Fig. 1.5. Sketch of process of obtaining depth information with Kinect (https://fabrizio89.wordpress.com/kinect-1-generation)

The Kinect sensor enables a wide variety of interactions, but it has "sweet spots" and limitations. Structured light approach poses challenges for objects that are located particularly near or particularly far from the sensor. For objects closer than 0.8 meters, the projected points appear too closely together for the sensor to measure; this results in a short-range blind spot that can have implications for where the sensor is mounted. For objects farther than four meters, the projected points fade into the background. This maximum range is also adversely affected by the presence of infrared noise in the environment. As such, the Kinect's range is significantly degraded outdoors. (*Mobile Manipulation for the KUKA YouBot Platform,* Worcester Polytechnic Institute)

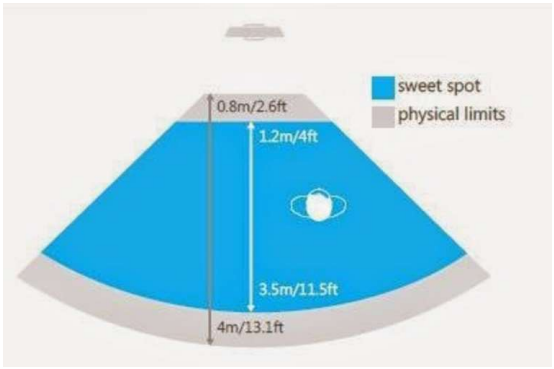


Fig. 1.6. Range of vision of Kinect (http://thethirdeye-idea.blogspot.lt)

The resulting projection of the environment is viewed by the integrated infrared camera and interpreted to produce depth information for each point. As a result, RGBD image is done, where each pixel has both a color and a distance, which can be used to map out body positions, gestures, motion or generate 3D maps. (*New trends in Medical and Service Robots*, Ecole Polytechnique Federale de Lausanne, Technical University of Cluj-Napoca, Institut Mihajlo Pupin)

1.3 Kuka YouBot

Kinect camera is incorporated to KUKA YouBot (Fig 1.7.). The KUKA YouBot is a mobile manipulator that was primarily developed for education and research by Kuka. It allows full control of all his sensors through open software. (*KUKA YouBot User Manual)*

Fig. 1.7. Kuka YouBot (http://www.expo21xx.com)

It is formed by the next elements:

- Mobile omnidirectional platform.
- 5-axes mobile arm and 2 finger gripper.

YouBot platform

YouBot platform (Fig.1.8.) uses an omnidirectional drive system with mecanum wheels (Fig. 1.9.). Mecanum wheels consist of a series of rollers mounted at a 45° angle. This allows the robot to move in any direction. Full specifications of KUKA YouBot platform are detailed in Annex B. (http://www.youbot-store.com)

Fig. 1.8. Kuka YouBot platform

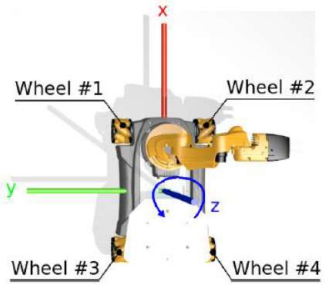


Fig. 1.9. Kuka YouBot wheels

YouBot Arm and gripper

YouBot Arm is a five degree-of-freedom arm. It is a five link serial kinematic chain with all revolute joints. The arm is similar to KUKA's larger industrial robot arms. The dimensions of each link of the arm, as well as the range of each joint are shown in Fig. 1.10. (*KUKA YouBot User Manual)*

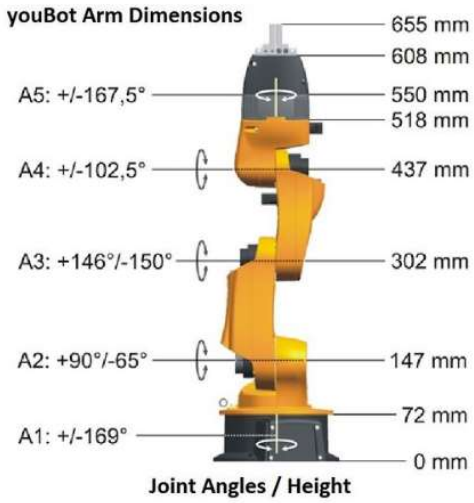


Fig. 1.10. Kuka YouBot Arm (http://www.youbot-store.com)

A relative encoder measures the rotation of each joint in the YouBot Arm. Users must manually move the arm to a home position before the arm is initialized. The scope of the movement of Kuka YouBot Arm is shown in Fig. 1.11. Full specifications of KUKA YouBot arm and gripper are detailed in Annex B. (*KUKA YouBot User Manual)*

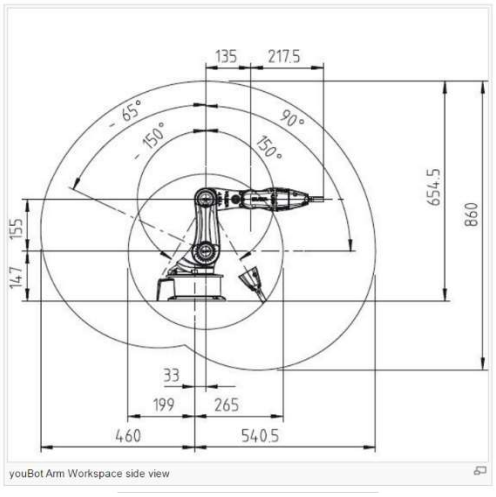


Fig. 1.11. Scope of movement of Kuka YouBot Arm (http://www.youbot-store.com)

2 THEORETICAL RESEARCH

In this chapter, it is expounded the theoretical research done. Firstly, there is a review of the software and the libraries that will be used. Secondly, it is explained the method that is implemented focusing on the theory that is behind. Finally, theoretical results are commented.

2.1 Software and Libraries

It is necessary to use different software already developed for solving the task. It has been decided to use Point Cloud Library for finding the centroid of the object, Robot Operating System for communicating with the robot, MoveIt! for planning its movements and Rviz for displaying the results.

2.1.1 Robot Operating System

The Robot Operating System (ROS) is an open source software framework for robotic development. ROS was originally developed starting in 2007 under the name Switchyard by the Stanford Artificial Intelligence Laboratory, but since 2008 it has been primarily developed by Willow Garage. ROS provides standard operating system services such as hardware abstraction, low-level device control, interprocess communication, and package management.

The main objective of ROS is to provide a common platform where developers can share and collaborate and make the construction of robotic applications quicker and easier. Actually, there are a lot of free open access libraries with ROS solutions (SLAM, motion planning, machine learning, etc.) that allows the developer to focus on his area of research without solving problems that are already solved. (*Robot Operating System*, https://en.wikipedia.org)

ROS works with Unix platforms and has been tried in systems as Linux and Mac OS X. In that project, it is used Ubuntu 12.04 and ROS version Hydro.

ROS is fundamentally a client / server system. It consists of a series of nodes (programs) that communicate with each other through the topics (broadcast) or services (interactive communication). There is a master node to which the nodes go to locate each other and obtain configuration information. In the next lines it is detailed its structure and how it works.

The different elements that compose the ROS system files are the next ones:

- **Packages:** are the main unit in the organization of the software. They can contain ROS processes, libraries, data, configuration files, etc.
- **Manifests**: XML files that contain the packages (license information, dependencies, etc.)
- **Stacks**: Collections of packages that contribute with specific functionalities.
- **Message types**: ROS message structures.
- **Service types**: service descriptions (define request and response structures).

The main elements of the functional structure of ROS and that compose the peer-to-peer net of processes are the next ones:

- **Nodes**: Processes, usually design with modularity. One system is composed by several nodes where each one has one specific functionality. The programming of the ROS nodes is done with ROS client libraries as roscpp (C++), rospy (Python) or roslisp (LISP).
- **Master:** process that works as main directory service. Master stores the information and the registration of topics and services and allows the nodes to interact between them. Nodes communicate with the Master for register information or request information of other nodes.
- **Messages**: are the data that nodes interchange.
- **Topics**: mechanism for broadcasting messages. One or multiple nodes can publish messages in one specific topic. Other nodes can subscribe to that topic and get the information.
- **Services:** mechanisms of communication of messages request-response type.
- **Bags:** format for store and reproduce messages.

In next lines, the process of connection to one topic is explained, due to it will be the same as the process will be done for obtain the point cloud information on this project.

1. It is initialized the subscriber. It reads the topic that it must subscribe.
2. It is initialized the publisher. It reads the topic that it must publish.
3. Subscriber registers on the Master.
4. Publisher registers on the Master.

5. Master informs the subscriber that there is a new publisher.
6. Subscriber contacts the publisher to request a new connection to the topic and the information of the transport protocol.
7. Publisher sends to the subscriber all the necessary data for connect to the chosen protocol.
8. Subscriber connects to the publisher and gets the data.

(*Taller de robótica y visión artificial para estudios de grado*, Universidad de Oviedo)

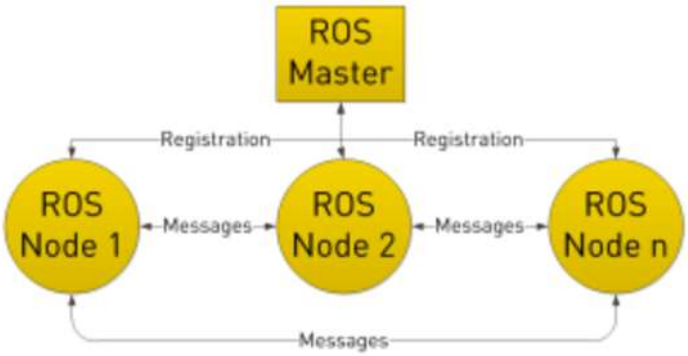


Fig. 2.1. Example of communication in ROS (*Intro to Ros*, http://www.clearpathrobotics.com)

2.1.2 Point Cloud Library

The Point Cloud Library (PCL) is an open source project for image and point cloud processing. PCL was originally developed by Willow Garage as a package for ROS. However, its utility made to convert it to a separate project. Nowadays, is maintained by the Open Perception Foundation. The PCL is split into multiple libraries which can be compiled and used separately. These libraries include support for: filtering, feature finding, key point finding, registration, kd-tree representation, octree representation, image segmentation, sample consensus, ranged images, file system I/O and visualization. (*Mobile Manipulation for the KUKA YouBot Platform,* Worcester Polytechnic Institute)

2.1.3 Rviz

Rviz (ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. Using Rviz, it is possible to visualize YouBot's current configuration on a virtual model of the robot. It is also possible to display live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more.

2.1.4 MoveIt!

MoveIt! is a motion planner framework. It incorporates the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It provides an easy-to-use platform for developing advanced robotics applications, evaluating new robot designs and building integrated robotics products for industrial, commercial and R&D. It is used on over 65 robots. In this project, Kuka YouBot's model for MoveIt! is used with YouBot Manipulation drivers developed by Sven Schneider.

In next lines the structure of MoveIt! and how it works will be explained.

In Fig. 2.2, the high-level system architecture for the primary node provided by MoveIt! called move_group is shown,. This node serves as an integrator, pulling all the individual components together to provide a set of ROS actions and services.

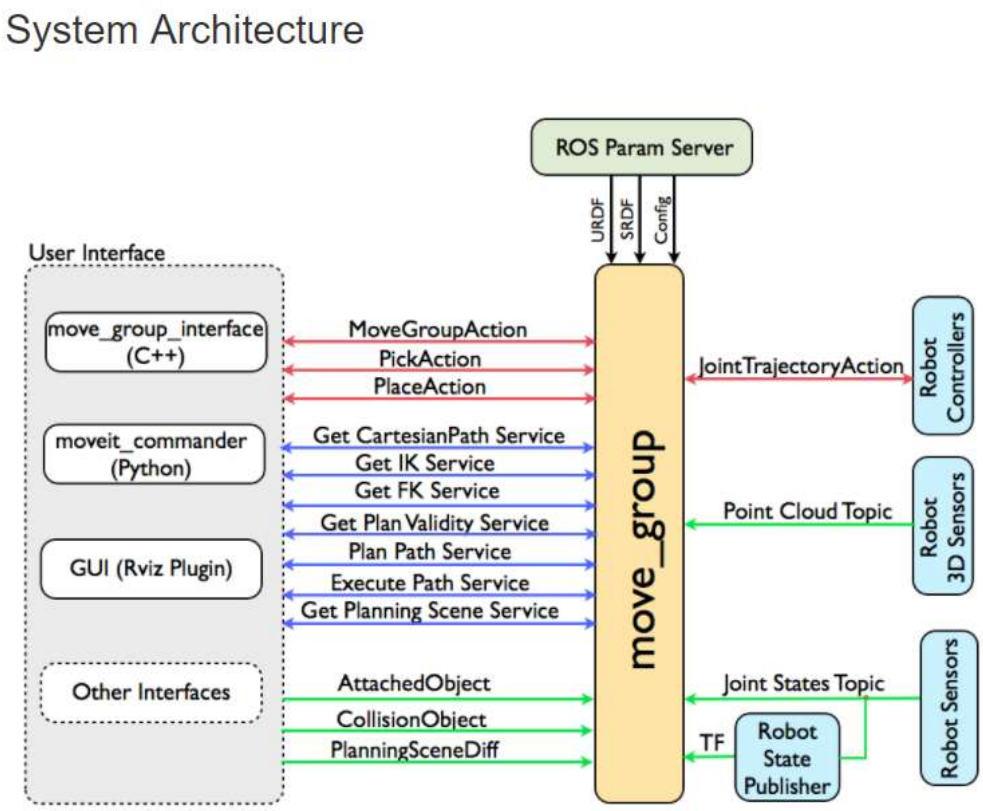


Fig. 2.2. MoveIt! Architecture (*Concepts*, http://moveit.ros.org)

Configuration

Move_group can be configured using the ROS param server. It will get three types of information:

- **URDF** (Universal Robotic Description Format): move_group looks for the robot_description parameter on the ROS param server to get the URDF for the robot.
- **SRDF** (Semantic Robot Description Format): move_group looks for the robot_description_semantic parameter on the ROS param server to get the SRDF for the robot. The SRDF is created by a user using the MoveIt! Setup Assistant.
- **MoveIt! configuration** - move_group will look on the ROS param server for other configuration specific to MoveIt! including joint limits, kinematics, motion planning, perception and other information. Config files for these components are automatically generated by the MoveIt! setup assistant and stored in the config directory of the corresponding MoveIt! config package for the robot.

Robot Interface

Move_group talks to the robot through ROS topics and actions. It communicates with the robot to get current state information (positions of the joints, etc.), to get point clouds and other sensor data from the robot sensors and to talk to the controllers on the robot.

Move_group monitors transform information using the ROS TF library. This allows the node to get global information about the robot's pose.

Move_group talks to the controllers on the robot using the FollowJointTrajectoryAction interface. This is a ROS action interface. A server on the robot needs to service this action - this server is not provided by move_group itself. Move_group will only instantiate a client to talk to this controller action server on the robot.

Move_group uses the Planning Scene Monitor to maintain a planning scene, which is a representation of the world and the current state of the robot. The robot state can include any objects carried by the robot which are considered to be rigidly attached to the robot.

Users interface

In order to communicate with the real robot, MoveIt! users have different options for access to the actions and services provided by move_group:

- Using move_group_interface package that provides a setup C++ interface to move_group.
- Using moveit_commander package with Python.
- Using Motion Planning plugin to Rviz.

In this project, it is tried last option, Motion Planning plugin to Rviz. In next lines, it is explained more detailed how it works.

Motion Planning

MoveIt! works with motion planners through a plugin interface. This allows MoveIt! to communicate with and use different motion planners from multiple libraries, making MoveIt! easily extensible. The interface to the motion planners is through a ROS Action or service (offered by the move_group node). The default motion planners for move_group are configured using OMPL and the MoveIt! interface to OMPL by the MoveIt! Setup Assistant.

The motion plan request clearly specifies what the user would like the motion planner to do. Usually, the user will ask the motion planner to move an arm to a different location (in joint space) or the end-effector to a new pose. Collisions are checked for by default (including self-collisions). It is possible to attach an object to the end-effector (or any part of the robot).

The move_group node will generate a desired trajectory in response to the motion plan request. This trajectory will move the arm (or any group of joints) to the desired location.

(*MoveIt! Concepts*, http://moveit.ros.org/documentation/concepts)

2.1.5 Other software

Besides the software that is already explained, this project is implemented using other drivers and libraries. The proposed method uses Kuka YouBot drivers provided by Kuka and some libraries as Eigen library or Standard Template Library.

2.2 Methodology

The method is implemented using C++ language programming and Ubuntu. In this chapter, method is explained step by step focusing on the theory that contains. It follows the next scheme:

1. Registration
2. Down sampling Point Cloud
3. Refer Kinect to Kuka YouBot coordinates
4. Get color information
5. Color transform to HSV space
6. Color filtering
7. Color transform to RGB space and pack color information
8. Normal estimation
9. Find centroid coordinates
10. Open Kuka YouBot model in MoveIt
11. Display RGBD point cloud from Kinect
12. Inverse Kinematics solver and movement planning
13. MoveIt Simulation
14. Execute movement in real robot

2.2.1 Registration

In Microsoft Kinect, depth and color images come from two separate, slightly offset, cameras. Depth data comes from IR sensors and RGB information from RGB camera. First step of the algorithm is to unite both information in one point cloud. For each pixel in the depth image, it is possible to calculate its position in 3D space and reproject it into the image plane of the RGB camera. In this way, it is possible to build up a *registered* depth image, where each pixel is aligned with its counterpart in the RGB image. (*Openni_launch*, http://wiki.ros.org)

In this project, registration is done using the registration capability provided by OpenNI library. It uses the Kinect factory calibration and it is possible to make a dynamic reconfiguration to change the ROS OpenNi driver's settings at runtime. It is done using the next command:

rosrun rqt_reconfigure rqt_reconfigure

As result of registration, it is possible to obtain depth-registered image (Fig. 2.5).

Fig. 2.3. Example Image RGB color

Fig. 2.4. Example Depth image

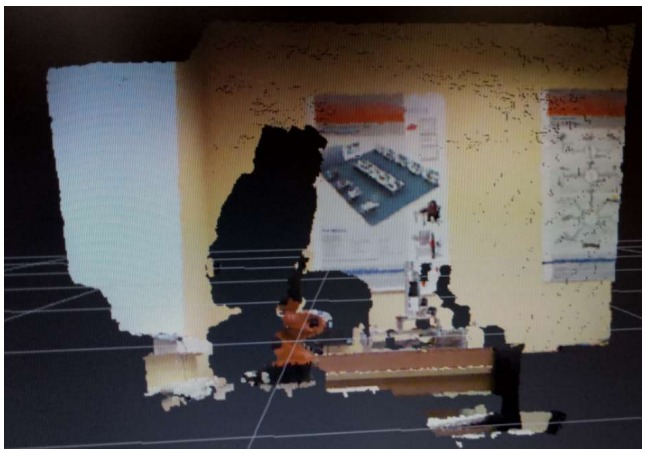


Fig. 2.5. Example depth-registered image

2.2.2 Down sampling Point Cloud

The second step on the algorithm is to minimize the data obtained with the Kinect. Normal point cloud can contain around 40000 points and it makes the algorithm work slowly. There are several different methods for simplifying the point cloud, in that project two of them are implemented: pass-through filter and voxel grid filter.

a) Pass-through filter

Pass-through filter implements a simple filtering along a specified dimension. It works cutting off values that are either inside or outside a given user range. This is really interesting due to the fact that in our experiment object will be located between some maximum ranges to be inside the Kinect view. Example of pass-through filter is shown in Fig. 2.6.:

```
pass_through_filter.setInputCloud (input);
pass_through_filter.setFilterFieldName ("z");
pass_through_filter.setFilterLimits (0, 1.5);
pass_through_filter.filter (*cloud_pass);
```

Fig. 2.6. Example pass-through filter

Firstly, algorithm reads the input cloud. Then it is indicated which axis will be filtered and the limits of the filter. Finally, all the points that are not between 0 and 1,5 meters in Z axis are eliminated. This process is repeated for X and Y axis with the following limits:

- Min value in X: -1.0
- Min value in Y: -1.0
- Min value in Z: 0
- Max value in X:1.0
- Max value in Y: 1.0
- Max value in Z: 1.5

It is important to clarify the axis coordinates of the Kinect (Fig. 2.7), for have clear the pass through filter range:

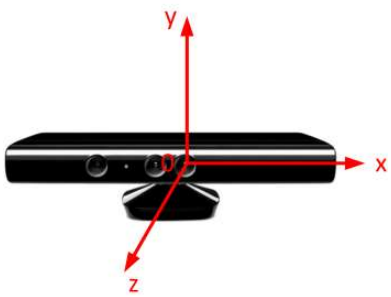


Fig. 2.7. Kinect axis coordinates (http://homes.cs.washington.edu)

Furthermore, this function not only eliminates the points that are outside the limits, it also eliminates Nan values.

b) Voxel grid filter

The VoxelGrid class creates a 3D voxel grid over the input point cloud data. Then, in each voxel, all the points present will be approximated with their centroid. This approach is a bit slower than approximating them with the center of the voxel, but it represents the underlying surface more accurately. (*Down sampling a point cloud using VoxelGrid*, http://pointclouds.org)

```
pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
sor.setInputCloud(cloud_pass_xyz);
sor.setLeafSize(0.001, 0.001, 0.001);
sor.filter(*cloud_filtered);
```

Fig. 2.8. Example Voxel grid

It is possible to variate the size of the voxel for adjust it to the experiment depending on the quantity of the data that programmer wants to down sample. In that project, size chosen is 0.001.

2.2.3 Refer Kinect to Kuka YouBot coordinates

At the moment of adding the Kinect camera on the simulation of Kuka YouBot model in Rviz, Kinect will be placed in the same position than Kuka YouBot base. Due to in every experiment, Kinect can be placed in different locations, the point cloud that displays has to be referred to the Kuka YouBot coordinates. In that project, Kinect will be located 0.3 meters at the right of Kuka YouBot and 0.15 meters behind. In that case, a simple translation of 0.3 meters in X axis and 0.15 in Z axis is applied. For that translation, Eigen library is used.

Eigen's Geometry module provides different kinds of geometric transformations as rotations, translations, scalings, etc. That permits to locate the Kinect in different positions and just manipulate the function for refer the point cloud data to the Kuka YouBot.

2.2.4 Get color information

Color information is available in the point cloud. For obtain it, a transformation from PCL Point Cloud to Point XYZRGB has to be done and save it as PCD file. It is obtained one field as it is shown in Fig. 2.9.

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 6393
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 6393
DATA ascii
-0.35605717 0.22741716 0.80400002 1.3272911e-38
-0.35605717 0.25881144 0.80400002 1.3827876e-38
-0.35452574 0.25881144 0.80400002 1.4102662e-38
-0.35376003 0.26110858 0.80400002 1.3550583e-38
-0.35376006 0.26340574 0.80400002 1.3550227e-38
-0.35299432 0.26570287 0.80400002 1.3552009e-38
-0.34840003 0.29480001 0.80400002 1.4191612e-38
```

Fig. 2.9. Example of PCD file with RGB color

PCD file obtained gives information about the Point Cloud. At is shown in Fig. 2.9, rgb information is packed into a float. It is possible to unpack it with the command in Fig. 2.10:

```
PointXYZRGB p;
// unpack rgb into r/g/b
uint32_t rgb = *reinterpret_cast<int*>(&p.rgb);
uint8_t r = (rgb >> 16) & 0x0000ff;
uint8_t g = (rgb >> 8)  & 0x0000ff;
uint8_t b = (rgb)       & 0x0000ff;
```

Fig. 2.10 Unpack RGB color (http://pointclouds.org)

In that project, it is not necessary to create a function to unpack the RGB information because a transformation to HSV color space is done and the unpacking of RGB information is included in the transformation function.

2.2.5 Color transform to HSV space

In this step, a transformation from RGB color space to HSV color space is done. HSV color space is a better representation to classify object color because it separates hue and saturation from value (lightness). Hue and saturation are often constant but lightness can change depending on the lighting strength in the room. Therefore, the detection criteria should be discriminative in the hue and saturation but invariant to lightness.

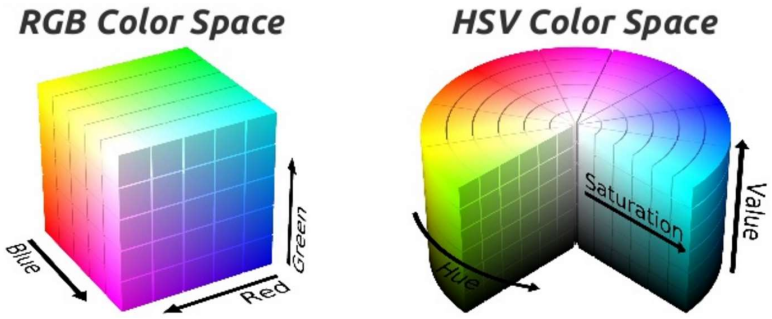


Fig. 2.11 RGB and HSV color space (*About Perception and Hue Histograms in HSV Space*)

The transformation of all the points in the point cloud from XYZRGB to XYZHSV is done following the function PointCloudXYZRGBtoXYZHSV from PCL library. This function transforms every point as shown in Fig. 2.12:

```
PointXYZRGBtoXYZHSV (PointXYZRGB& in,
                     PointXYZHSV& out)
{
  const unsigned char max = std::max (in.r, std::max (in.g, in.b));
  const unsigned char min = std::min (in.r, std::min (in.g, in.b));

  out.v = static_cast <float> (max) / 255.f;

  if (max == 0) // division by zero
  {
    out.s = 0.f;
    out.h = 0.f; // h = -1.f;
    return;
  }

  const float diff = static_cast <float> (max - min);
  out.s = diff / static_cast <float> (max);

  if (min == max) // diff == 0 -> division by zero
  {
    out.h = 0;
    return;
  }

  if      (max == in.r) out.h = 60.f * (      static_cast <float> (in.g - in.b) / diff);
  else if (max == in.g) out.h = 60.f * (2.f + static_cast <float> (in.b - in.r) / diff);
  else                  out.h = 60.f * (4.f + static_cast <float> (in.r - in.g) / diff); // max == b

  if (out.h < 0.f) out.h += 360.f;
}
```

Fig. 2.12. Transformation of PointXYZRGB to PointXYZHSV (http://pointclouds.org)

2.2.6 Color filtering

Color filter is applied to exclude the points that are not part of the red object. There are several types of filters, the filter used in that project is a Pass-through filter similar to the one implemented in point 2.2.2. In that case, it is filtered in hue and saturation values due to the reasons explained in point 2.2.5. The maximum and minimum values can be chosen experimentally, adapting the limits to the object color that will be segmented. In that case, the minimum and maximum values chosen for the red object are the next ones:

- Hue minimum value: 300
- Hue maximum value: 360
- Saturation minimum value: 0.2
- Saturation maximum value: 1

2.2.7 Color transformation to RGB space and pack color information

After filtering, color transformation to RGB space is done for continue working with PCL. In that case, any function is provided by PCL. Function below (Fig 2.13.), has been created and added to the library: point_types_conversion.h

```
PointCloudXYZHSVtoXYZRGB (const PointCloud<PointXYZHSV>& in,
                          PointCloud<PointXYZRGB>& out)
{
    out.width = in.width;
    out.height = in.height;
    for (size_t i = 0; i < in.points.size (); i++)
    {
      PointXYZRGB p;
      PointXYZHSVtoXYZRGB (in.points[i], p);
      out.points.push_back(p);
    }
}
```

Fig. 2.13. PointCloudXYZHSVtoXYZRGB (http://pointclouds.org)

It uses function PointXYZHSVtoXYZRGB (Fig. 2.14.).

```
PointXYZHSVtoXYZRGB (PointXYZHSV&  in,
                     PointXYZRGB&  out)
{
  out.x = in.x; out.y = in.y; out.z = in.z;
  if (in.s == 0)
  {
    out.r = out.g = out.b = static_cast<uint8_t> (in.v);
    return;
  }
  float a = in.h / 60;
  int   i = static_cast<int> (floorf (a));
  float f = a - static_cast<float> (i);
  float p = in.v * (1 - in.s);
  float q = in.v * (1 - in.s * f);
  float t = in.v * (1 - in.s * (1 - f));

  switch (i)
  {
    case 0:
    {
      out.r = static_cast<uint8_t> (255 * in.v);
      out.g = static_cast<uint8_t> (255 * t);
      out.b = static_cast<uint8_t> (255 * p);
      break;
    }
    case 1:
    {
      out.r = static_cast<uint8_t> (255 * q);
      out.g = static_cast<uint8_t> (255 * in.v);
      out.b = static_cast<uint8_t> (255 * p);
      break;
    }
    case 2:
    {
      out.r = static_cast<uint8_t> (255 * p);
      out.g = static_cast<uint8_t> (255 * in.v);
      out.b = static_cast<uint8_t> (255 * t);
      break;
    }
    case 3:
    {
      out.r = static_cast<uint8_t> (255 * p);
      out.g = static_cast<uint8_t> (255 * q);
      out.b = static_cast<uint8_t> (255 * in.v);
      break;
    }
    case 4:
    {
      out.r = static_cast<uint8_t> (255 * t);
      out.g = static_cast<uint8_t> (255 * p);
      out.b = static_cast<uint8_t> (255 * in.v);
      break;
    }
    default:
    {
      out.r = static_cast<uint8_t> (255 * in.v);
      out.g = static_cast<uint8_t> (255 * p);
      out.b = static_cast<uint8_t> (255 * q);
      break;
    }
  }
}
```

Fig. 2.14. Function PointXYZHSVtoPointXYZRGB (http://pointclouds.org)

2.2.8 Normal estimation

Normal estimation of all the points is done for later find the 3D centroid of the point cloud. It is done using KdTree search method. Point cloud data is finally concatenate in XYRZGBNormal pcd file.

2.2.9 Find centroid

The centroid is found using the function compute3DCentroid (Fig. 2.15.) available in PCL library.

```
pcl::compute3DCentroid (const pcl::PointCloud<PointT> &cloud
                        Eigen::Matrix<Scalar, 4, 1> &centroi
{
  if (cloud.empty ())
    return (0);

  // Initialize to 0
  centroid.setZero ();
  // For each point in the cloud
  // If the data is dense, we don't need to check for NaN
  if (cloud.is_dense)
  {
    for (size_t i = 0; i < cloud.size (); ++i)
    {
      centroid[0] += cloud[i].x;
      centroid[1] += cloud[i].y;
      centroid[2] += cloud[i].z;
    }
    centroid[3] = 0;
    centroid /= static_cast<Scalar> (cloud.size ());

    return (static_cast<unsigned int> (cloud.size ()));
  }
  // NaN or Inf values could exist => check for them
  else
  {
    unsigned cp = 0;
    for (size_t i = 0; i < cloud.size (); ++i)
    {
      // Check if the point is invalid
      if (!isFinite (cloud [i]))
        continue;

      centroid[0] += cloud[i].x;
      centroid[1] += cloud[i].y;
      centroid[2] += cloud[i].z;
      ++cp;
    }
    centroid[3] = 0;
    centroid /= static_cast<Scalar> (cp);

    return (cp);
  }
}
```

Fig. 2.15. Function compute3Dcentroid (http://pointclouds.org)

The centroid that will be found is the centroid of the point cloud, that is not going to be the same that the centroid of the object. For obtain, the centroid of the object some model has to be implemented. For example, RANSAC modelling available in PCL is a good option for model spheres or cylinders. In that project, it is not tested because objects that will be tried have different forms.

2.2.10 Load Kuka Youbot model in MoveIt!

First, Kuka Youbot is initialized. After that, Kuka Youbot model (Fig. 2.16.) in MoveIt! is load using Sven Schneider driver call it Youbot_manipulator. The youbot_manipulation meta-package contains the following packages:

- **youbot_arm_kinematics**: A framework-independent library which implements an analytical inverse position kinematics solver for the youBot manipulator.
- **youbot_arm_kinematics_moveit**: A MoveIt! plugin which exposes the youbot_arm_kinematics library to MoveIt!.
- **youbot_moveit**: Configuration files to use the youBot with MoveIt! and the analytical inverse kinematics solver from this repository.

(*Youbot_manipulator*, https://github.com/svenschneider/youbot-manipulation)

With the next command line it is possible to run MoveIt! and visualize in Rviz the Youbot model:

Roslaunch youbot: roslaunch youbot_moveit demo.launch

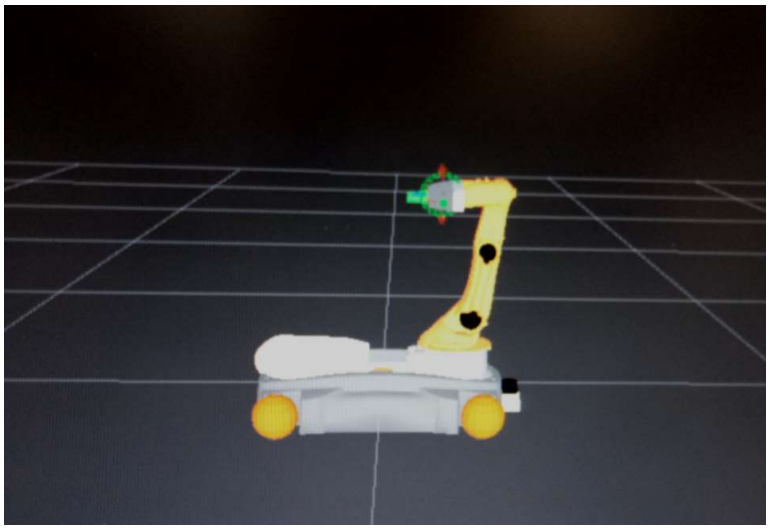


Fig. 2.16. Kuka Youbot Model in MoveIt!.

Next step, it is to add the camera and the point cloud data with the registered points to the Kuka YouBot's workspace.

2.2.11 Inverse Kinematics solver and motion planning

After MoveIt! is initialized and Point Cloud data is displayed into the map, it is the moment to move the YouBot arm in the model to the final desired position. In Rviz is possible to visualize the initial state and the final state. In that case initial state is with the arm folded in home position and final state is where the object is located.

Once the arm has been located in the desired final position, it is possible to plan the movement with motion planning. YouBot Manipulator driver includes an inverse kinematics solver for the Kuka YouBot.

Furthermore, Motion planning includes an option for avoiding collisions, with the robot itself or with the outside world.

2.2.12 MoveIt Simulation

When the Motion planning is done, simulation of the movement of the robot is displayed in MoveIt!. It is possible to display it in normal loop or with a trail that marks all the positions where Kuka YouBot goes over.

2.2.13 Execute movement in real robot

For executing the movement in the real robot, MoveIt! users have different options for access to the actions and services provided by move_group (Fig 2.2.): using move_group_interface package (C++ interface), using moveit_commander package (python) or using Motion planning plugin with Rviz. In that project, Motion planning plugin in Rviz is tried.

2.3 Theoretical conclusions

The result of the theoretical research is the algorithm developed shown in Annex A. It will be tested in the experimental, however it is possible to extract some conclusions of the theoretical research before try the algorithm:

- It has to be done a registration process for unite image and depth information obtained by Kinect.
- Down sampling of the point cloud is necessary to reduce the number of points and make the algorithm work faster.
- Transformation from RGB to HSV color space is done for avoid lightness effect.
- Centroid of the point cloud will be found. It will not be exactly the same as the centroid of the object due to any modelling (as RANSAC modelling) is applied.
- It is possible to integrate Kinect 3D sensor in MoveIt! with Kuka YouBot model.
- MoveIt! is a useful tool for motion planning. It is possible to simulate the movement of the robot avoiding collisions.

3 EXPERIMENTAL RESEARCH

3.1 Experiment

The experiment consists in try to pick up two different red objects detected with Kinect: one red pencil case (Fig. 3.1.) and one tomato (Fig. 3.2.).

Fig. 3.1. Red pencil case

Fig. 3.2. Red tomato

They have been located in the same position (0.3, 0, 0.8) from the Kuka YouBot reference. Kinect camera has been located near the Kuka YouBot, 0.3 meters on the right of the YouBot and 0.15 meters behind (Fig. 3.3.)

Fig. 3.3. Position of Kinect camera and Kuka YouBot

Results will be analyzed dividing the experiment in three different parts. Firstly, it will be evaluated the quality of the object detection, comparing the location of the objects with the position of the centroid that is found. After that, it will be examined if it is possible to plan the movement of the robot without any obstacles. Finally, it will be analyzed if the movement of the robot is the same that the planned by MoveIt!

3.2 Results

In this chapter, results that are obtained in every step of the method are explained. Firstly, the RGB images obtained with Kinect are shown in Fig. 3.4 and Fig. 3.5:

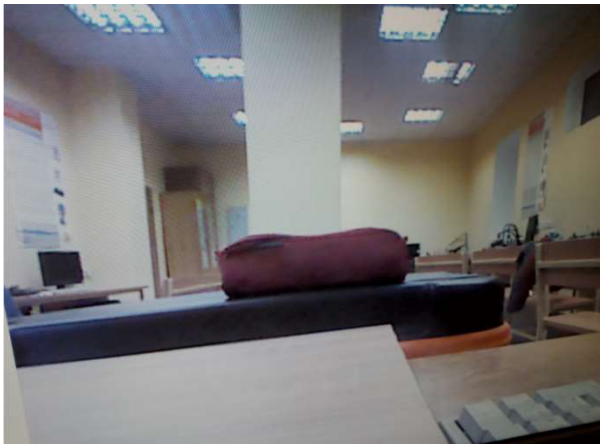


Fig. 3.4. RGB image pencil case

Fig. 3.5. RGB image tomato

The depth images obtained with Kinect are shown in Fig. s 33 and 34:

Fig. 3.6. Depth image pencil case

Fig. 3.7. Depth image tomato

The first step of the method is the registration and visualization in Rviz. It is possible to appreciate how pencil case is perfectly registered (Fig.3.8.). Nevertheless, the tomato is not registered correctly. In some moments, red parts of the tomato appear (Fig. 3.9.), but in other moments tomato does not appear on the image (Fig.3.10.).

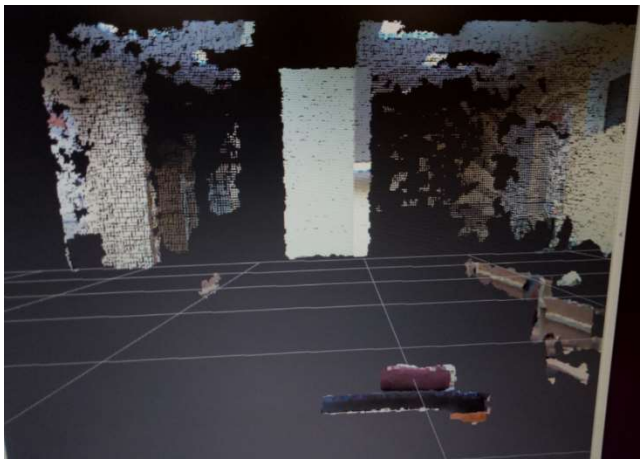


Fig. 3.8. Registered point cloud image pencil case

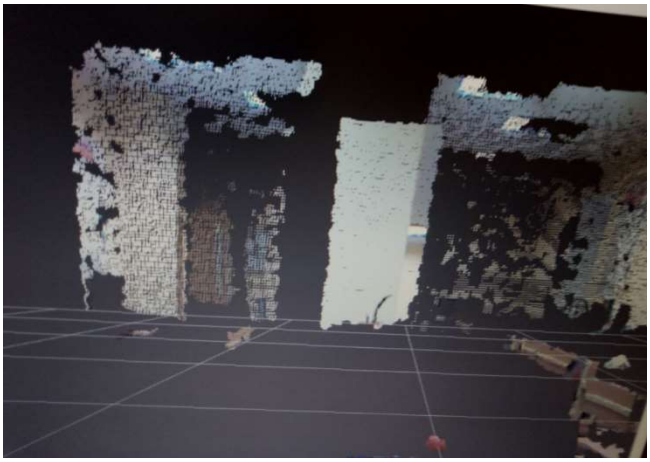


Fig. 3.9. Registered point cloud image tomato appear

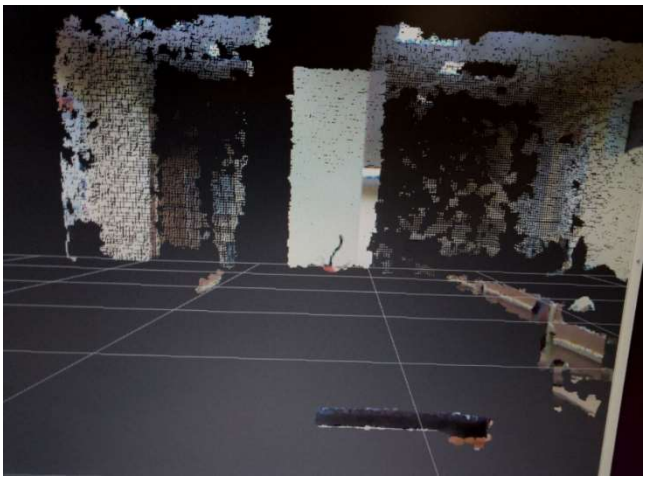


Fig. 3.10. Registered point cloud image tomato not appear

The experiment has continued using a point cloud where tomato appears.

After that, down sampling and acquisition of RGB information is done. As result, is obtained a Point Cloud Data with 28152 points for pencil case (original had 462432 points) (Fig. 3.11.) and 19234 points for tomato (Fig 3.12.). It is possible to visualize that pcd files converting them to Point Cloud with next command:

Rosrun pcl_ros pcd_to_pointcloud xyzrgb.pcd

Pcl_viewer xyzrgb.pcd

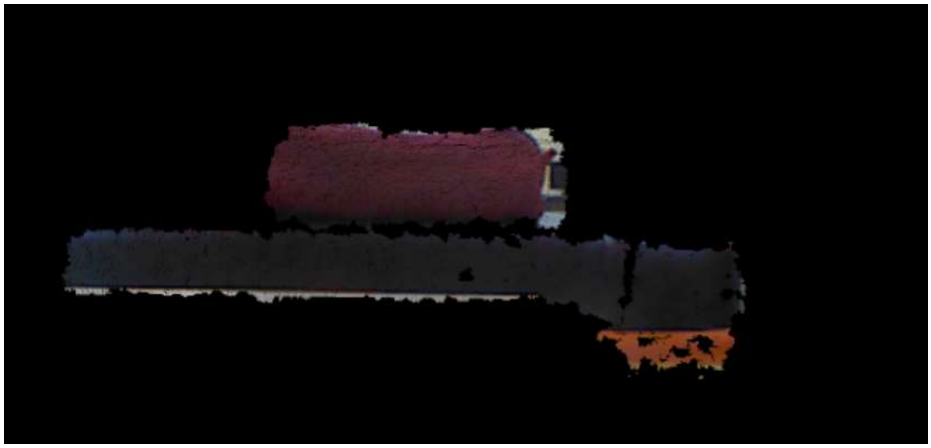


Fig. 3.11. XYZRGB point cloud pencil case

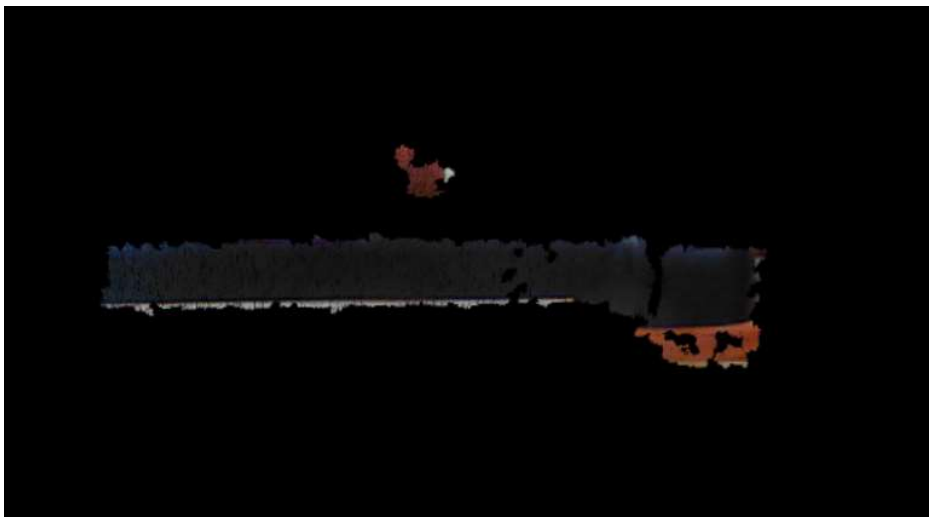


Fig. 3.12. XYZRGB point cloud tomato

Then transformation to HSV color space is done. Point cloud data that result of this transformation (Fig. 3.13.). It contains the same number of points that XYZRGB point cloud.

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z h s v
SIZE 4 4 4 4 4 4
TYPE F F F F F F
COUNT 1 1 1 1 1 1
WIDTH 28152
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 28152
DATA ascii
-0.35361236 0.12659144 0.047000051 27.272728 0.23404256 0.5529412
-0.35361236 0.12763335 0.047000051 20.930233 0.31159419 0.5411765
-0.4147743 0.11659715 0.049000025 14.25 0.63999999 0.49019608
-0.41372856 0.11659715 0.049000025 14.814815 0.648 0.49019608
-0.41059142 0.11659715 0.049000025 14.117647 0.67460316 0.49411765
-0.41686571 0.11764286 0.049000025 13.6 0.61475408 0.47843137
```

Fig. 3.13. Point Cloud Data XYZHSV format pencil case

Point clouds are visualized following the same process than XYZRGB point clouds. Results are shown in Fig. 3.14. and Fig. 3.15.

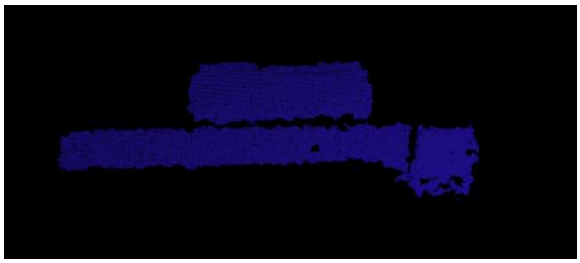


Fig. 3.14. XYZHSV point cloud pencil case

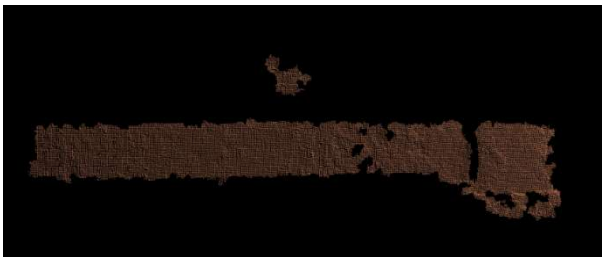


Fig. 3.15. XYZHSV point cloud tomato

Color red filter is applied. Now, point clouds contain 7704 Points for the pencil case (Fig. 3.16.) and 358 points for the tomato (Fig.3.17.).

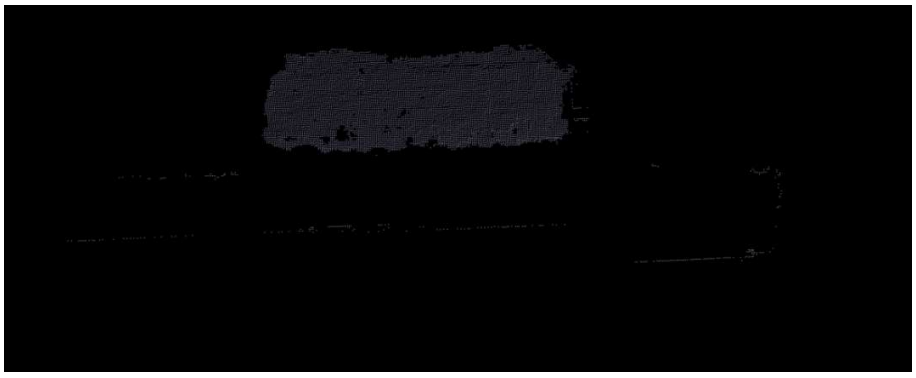


Fig. 3.16. Pencil case point cloud after red filter

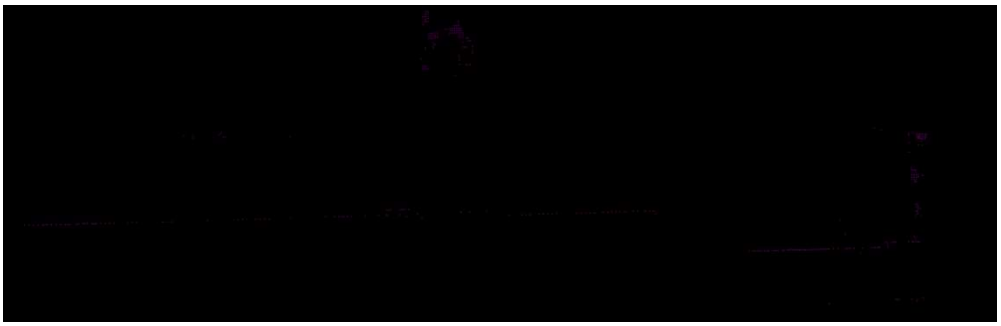


Fig. 3.17. Tomato point cloud after red filter

Due to the bad result with the tomato point cloud after red filter, the experiment continues only with pencil case.

Then, color transform to RGB and normal estimation is done. Point Cloud data obtained has this format:

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z rgb normal_x normal_y normal_z curvature
SIZE 4 4 4 4 4 4 4 4
TYPE F F F F F F F F
COUNT 1 1 1 1 1 1 1 1
WIDTH 7704
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 7704
DATA ascii
-0.35254189 0.12436858 0.050999999 6.9006354e-39 -0.020100549 -0.3623625 -0.93182045 0.0022828274
-0.35200667 0.12376668 0.053000033 8.4639955e-39 -0.020100033 -0.36236584 -0.9318192 0.0022825424
-0.36254001 0.12482001 0.053000033 7.8225903e-39 -0.0028100619 -0.39180675 -0.92004323 0.0016024505
-0.35068381 0.12399048 0.05400002 8.2799629e-39 -0.020100033 -0.36236584 -0.9318192 0.0022825424
-0.35596001 0.12504572 0.05400002 7.8240253e-39 -0.020099981 -0.36236608 -0.93181908 0.0022820437
-0.3409 0.12527144 0.055000007 5.7986193e-39 -0.020100033 -0.36236584 -0.9318192 0.0022825424
```

Fig. 3.18. Point Cloud data XYZRGBNormal format pencil case

Centroid is found and result is shown on the terminal screen (Fig. 3.19.).

```
The XYZ coordinates of the centroid are: (0.306545, 0.00625121, 0.748213).
The XYZ coordinates of the centroid are: (0.304263, 0.00599307, 0.747681).
The XYZ coordinates of the centroid are: (0.305273, 0.00645464, 0.747874).
The XYZ coordinates of the centroid are: (0.305285, 0.00631175, 0.747748).
```

Fig. 3.19. Centroid displayed on screen for pencil case

After that, Rviz and MoveIt! are initialized and YouBot model is upload. Kinect camera is added and it is possible to visualize registered image at the same time than Kuka YouBot Model (Fig. 3.20.).

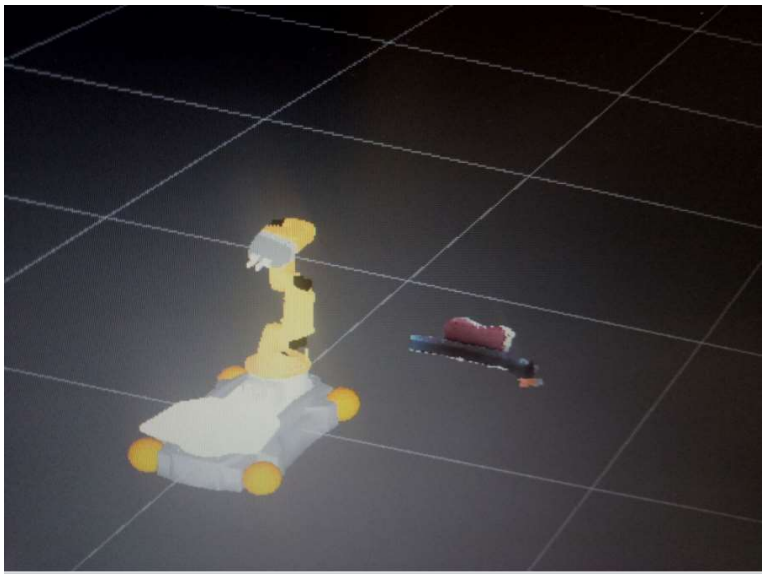


Fig. 3.20. Kuka YouBot Model and registered image pencil case

Then YouBot Model is manipulated graphically for establishing the goal position the same where the object is located (Fig. 3.21.). As initial position, it is chosen folded position. Planning of motion control is done and inverse kinematics are solved. As result, it is possible to perfectly reproduce a simulation for moving the YouBot to goal position.

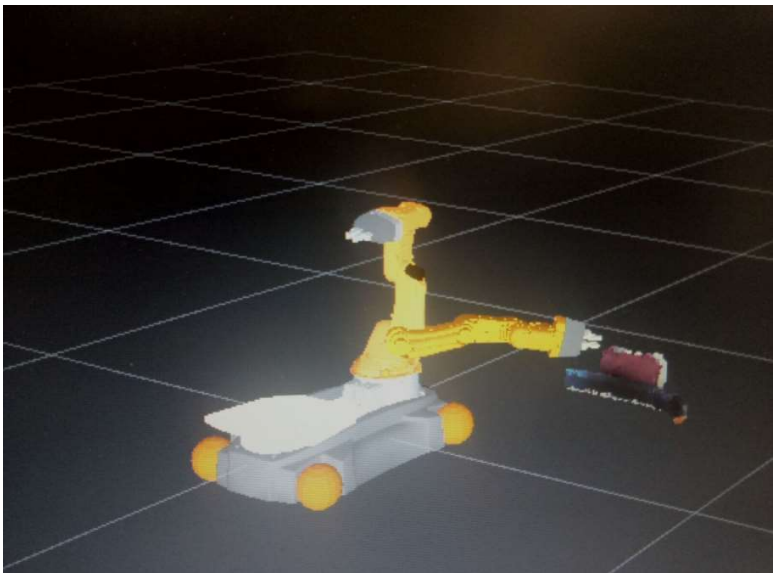


Fig. 3.21. Kuka YouBot Model final position

In Fig. 3.22., it is shown the trail that Kuka YouBot model follows for arrive to the desired position.

Fig. 3.22. Trail simulation movement of the YouBot.

Finally, execution in real robot is done. Kuka YouBot starts to move following the MoveIt! trajectory but since the first moment starts to overshoot, it has not arrived to the desired position and control of the robot is lost.

3.3 Evaluation of results

3.3.1 Find centroid

For the pencil case, the process of detection, registration and color filtering has been done correctly. The centroid of the point cloud data is well obtained. Comparing with the point where is situated the pencil case, there is just a small variation in Z axis due to the algorithm just compute the points that Kinect can see and any 3D modeling of the pencil case is done. In future works, a cylinder RANSAC modeling could be applied for finding the exact center of mass.

For the tomato, not all the points appear in the registered image and on some occasions any of them. That is produced because of the automatic registration process provided by Kinect is not good at all with sphere objects as the tomato. Then, the obtained points after the color filtering are not enough for finding the centroid properly. Even with that problem with registration, color filter should be improved to adapt the values to the tomato red color.

3.3.2 Motion planning

The motion planning tried with MoveIt! works good. It solves the inverse kinematics and it plans the motion of the Kuka YouBot perfectly, avoiding the possible collisions with itself. The simulation is displayed correctly.

3.3.3 Execution in real robot

Problem appear with the real robot that starts to overshoot. After consulting more documentation, it is arrived to the conclusion that the problem is that trajectory from MoveIt! uses Trajectory messages and require torque controller which for Kuka YouBot has not been configured in YouBot drivers.

There are different approaches to try to solve this problem, but it has not been completely investigated. First approach is to try to tune the parameters of the configuration of the YouBot experimentally. Second approach, and more valid than first one is to convert the geometry messages or trajectory messages into arm_controller/position message. Finally, third approach is to incorporate a Torque controller as is done in thesis: "*Torque Control of a KUKA youBot Arm*".

CONCLUSIONS

Importance of robots in nowadays industry makes this project and other researches in mechatronics and robotics really important. In this paper, Kinect 3D camera is integrated in Kuka YouBot and the development of an algorithm to show its capabilities is done. Results have shown that Kinect can provide vision to Kuka for recognition and detection of objects. Furthermore, integration of the Kinect in Kuka YouBot's model world in MoveIt! is done. It is shown that MoveIt! is a really useful software for planning motion trajectories and simulating them. Finally, to move the robot with Rviz pluguin in MoveIt! has been tried and some problems have appeared. As it commented before, the method can be improved to obtain better results. In this chapter, there is an enumeration of different future developments that should be done for making the method more robust.

The first idea is related to the process of the object detection. In future, it should be integrated a second camera for having a better vision of the objects. With a second camera, it will be possible to avoid problems in registration as happened to tomato. Furthermore, model the objects and find the real 3D centroid of the object, not only the centroid of the point cloud provide by the method. Another possibility, but probably less consistent, is to try a different registration process, not the one that Kinect driver provides.

Related to improving object detection, in algorithm is shown how to do a RANSAC modeling for spheres. It has not been applied to the project because the tomato was not detected properly, but should be implemented in the future when spheres will be well registered.

Another possible improvement is in the centroid detection, incorporate a Eucledian cluster for avoiding the outline points.

Focusing on the Motion planning, it should be incorporated into the model world a base as real robot has in the laboratory for avoiding possible collisions. Furthermore, the coordinates of the centroid should be published as a topic and a program should be created to read this topic and add the coordinates as goal state position in MoveIt motion planner.

Finally, it should be solved the problems with the overshooting of the real robot and improve the integration in MoveIt! as is commented in point 3.3.3. A possible solution is to incorporate a Torque control to the YouBot driver.

ACKNOWLEDGMENTS

Personally, to work in that project has increase exponentially my knowledge in robotics and programming. I have learnt how ROS works and how to use different libraries already developed for creating my own algorithm. Furthermore, I have learn how to contribute to the developing and research in robotics using GitHub or other online platforms.

I am very grateful to my mentor on this thesis, Assoc. Prof. Dr. Vytautas Bučinskas for give me the opportunity to work with him and for all his advices and recommendations.

As well to the Vilniaus Gedimino Technikos Universitetas in general and to the Department of Mechatronics and Robotics for let me study and learn in this institution.

Finally, thanks to ESN VGTU and all friends I have met in Lithuania for help me to adapt to Vilnius and let me feel like at home.

REFERENCES

Alves M., *About Perception and Hue Histograms in HSV Space*, University of Rio de Janeiro, Brasil.

Avizzano C., Brizi F., Peppoloni L., Ruffaldis E., *Immersive, ROS-integrated Framework for Robot Teleoperation,* PERCRO Laboratory, Tecip Institute Scuola Superiore Sant'Anna

Baker, D.; Wampler, C.: *On the Inverse Kinematics of Redundant Manipulators*, International Journal of Robotics Research, vol.7, No.2, 1988

Bo L., Fox D., Lai K., Ren X., *Sparse distance learning for object recognition combining RGB and depth information*, in: 2011 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2011, pp. 4007–4013.doi:10.1109/ICRA.2011.5980377

Berenson, D.; Srinivasa, S.; Ferguson, D.; Romea, A.; Kuffner, J.: *Manipulation planning with workspace goal regions*, Proc. of the IEEE International Conference of Robotics and Automation (ICRA-2009), IEEE Press, 1999.

Bleuler H., Pisla A., Pisla D.,Rodic A., Vaida C., *New trends in Medical and Service Robots*, Ecole Polytechnique Federale de Lausanne, Switzerland Technical University of Cluj-Napoca, Romania, Institut Mihajlo Pupin, Serbia, 2014.

Bolles C., Fischler M., *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*,

Charkivo R., Dwiputra R., Prassler E., Zakharov A., *Model for the youBot Manipulator* Bonn-Rhein-Sieg University of Applied Sciences, Department of Computer Science Grantham-Allee, Sankt Augustin, Germany

Craig, J.: *Introduction to Robotics Mechanics and Control*, Third edition, Chapter 3, Manipulator Kinematics, Pearson Prentice Hall, New Jersey, USA, 2005.

Cremers D., Steinbruecker F., Sturm J., *Real-time visual odometry from dense rgb-d images*, in: Workshop on Live Dense Reconstruction with Moving Cameras at the Intl. Conf. on Computer Vision (ICCV), 2011.

Fox D., Henry P., Herst E., Krainin M., Rgbd mapping: *Using depth cameras for dense 3d modeling of indoor environments, in: In RGB-D*: Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS, 2010

Hlaváč V., *Sensors for Robots*, Czech Technical University, Prague, Czech Republic

Jenkel T., Kelly R., Shepanski P., *Mobile Manipulation for the KUKA youBot Platform,* Worcester Polytechnich Institute, USA, 2013

Keiser B., *Torque Control of a KUKA youBot Arm*, Master thesis, Robotics and Perception Group University of Zurich, 2013

Khatib, O. Siciliano, B., editors: *Springer Handbook of Robotics*, Springer, Berlin, Heidelberg, Germany, 2008.

KUKA, *KUKA YouBot User Manual*, 2012

Microsoft, "*Kinect Sensor*," 2013. http://msdn.microsoft.com/enus/library/hh438998.aspx

MoveIt! Motion Planning Framework, *MoveIt! Tutorials*, http://docs.ros.org/indigo/api/moveit_tutorials/html/

Point Cloud Library, *Down sampling a point cloud using VoxelGrid*, http://pointclouds.org

Sánchez F., *Taller de robótica y visión artificial para estudios de grado*, Centro internacional de Posgrado, Universidad de Oviedo, 2013

ANNEX

ANNEX A. ALGORITHM

```
#include <ros/ros.h>
#include <iostream>
// PCL specific includes
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/io/pcd_io.h>
#include <pcl/sample_consensus/ransac.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/PCLPointCloud2.h>
#include <pcl_ros/transforms.h>
#include <pcl/point_types_conversion.h>
#include <pcl/features/normal_3d.h>

ros::Publisher pub;

void cloud_cb(const pcl::PCLPointCloud2ConstPtr& input)
{

//pass through filter

  pcl::PCLPointCloud2::Ptr cloud_pass(new pcl::PCLPointCloud2);
  pcl::PCLPointCloud2::Ptr cloud_pass_xz(new pcl::PCLPointCloud2);
  pcl::PCLPointCloud2::Ptr cloud_pass_xyz(new pcl::PCLPointCloud2);
  pcl::PassThrough<pcl::PCLPointCloud2> pass_through_filter;

    pass_through_filter.setInputCloud (input);
    pass_through_filter.setFilterFieldName ("z");
    pass_through_filter.setFilterLimits (0, 1.5);
    pass_through_filter.filter (*cloud_pass);


    pass_through_filter.setInputCloud (cloud_pass);
    pass_through_filter.setFilterFieldName ("x");
    pass_through_filter.setFilterLimits (-1.0, 1.0);
    pass_through_filter.filter (*cloud_pass_xz);

    pass_through_filter.setInputCloud (cloud_pass_xz);
    pass_through_filter.setFilterFieldName ("y");
    pass_through_filter.setFilterLimits (-1.0, 1.0);
    pass_through_filter.filter (*cloud_pass_xyz);


// voxel grid filter

  pcl::PCLPointCloud2::Ptr cloud_filtered (new pcl::PCLPointCloud2);

    pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
    sor.setInputCloud(cloud_pass_xyz);
    sor.setLeafSize(0.001, 0.001, 0.001);
    sor.filter(*cloud_filtered);
```

```
// Transform PCLPointcloud2 in Pointcloud XYZRGB;

  pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_xyzrgb(new pcl::PointCloud<pcl::PointXYZRGB>);
  pcl::fromPCLPointCloud2(*cloud_filtered,*cloud_xyzrgb);

  pcl::io::savePCDFileASCII ("xyzrgb.pcd", *cloud_xyzrgb);

// Define a translation of 1 meters on the x axis.
  Eigen::Affine3f transform_2 = Eigen::Affine3f::Identity();
  transform_2.translation() << 0.3, 0.0, 0.15;

  // Executing the transformation
  pcl::PointCloud<pcl::PointXYZRGB>::Ptr translated_xyzrgb (new pcl::PointCloud<pcl::PointXYZRGB> ());
  pcl::transformPointCloud (*cloud_xyzrgb, *translated_xyzrgb, transform_2);

   // Convert to publish

  pcl::PCLPointCloud2::Ptr cloud_publish (new pcl::PCLPointCloud2);
  pcl::toPCLPointCloud2(*translated_xyzrgb,*cloud_publish);

  pcl::PointCloud<pcl::PointXYZRGB>::Ptr translated_xyzrgb2(new pcl::PointCloud<pcl::PointXYZRGB>);
  pcl::fromPCLPointCloud2(*cloud_publish,*translated_xyzrgb2);

// Color transform POINTXYZRGBtoXYZHSV
  pcl::PointCloud<pcl::PointXYZHSV>::Ptr cloud_xyzhsv(new pcl::PointCloud<pcl::PointXYZHSV>);
  pcl::PointCloudXYZRGBtoXYZHSV(*translated_xyzrgb, *cloud_xyzhsv);

   pcl::io::savePCDFileASCII ("xyzhsv.pcd", *cloud_xyzhsv);
   pcl::PointCloud<pcl::PointXYZHSV>::Ptr cloud_color_filtered_h (new pcl::PointCloud<pcl::PointXYZHSV>);
   pcl::PointCloud<pcl::PointXYZHSV>::Ptr cloud_color_filtered_hv (new pcl::PointCloud<pcl::PointXYZHSV>);

   pcl::PassThrough<pcl::PointXYZHSV> pass;
       pass.setInputCloud (cloud_xyzhsv);
       pass.setFilterFieldName ("h");
       pass.setFilterLimits (300, 360);
       //pass.setFilterLimitsNegative (true);
       pass.filter (*cloud_color_filtered_h);
```

```
   pcl::io::savePCDFileASCII ("cloud_color_filtered_h.pcd", *cloud_color_filtered_h);

       pass.setInputCloud (cloud_color_filtered_h);
       pass.setFilterFieldName ("s");
       pass.setFilterLimits (0.2, 1);
       //pass.setFilterLimitsNegative (true);
       pass.filter (*cloud_color_filtered_hv);

   /*   pass.setInputCloud (cloud_color_filtered_h);
       pass.setFilterFieldName ("v");
       pass.setFilterLimits (0.2, 0.4);
       //pass.setFilterLimitsNegative (true);
       pass.filter (*cloud_color_filtered_hv);
*/
   pcl::io::savePCDFileASCII ("cloud_color_filtered_hv.pcd", *cloud_color_filtered_hv);

// Color transform PointXYZHSVtoXYZRGB

    pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_color_rgb(new pcl::PointCloud<pcl::PointXYZRGB>);
    pcl::PointCloudXYZHSVtoXYZRGB(*cloud_color_filtered_hv, *cloud_color_rgb);

    pcl::io::savePCDFileASCII ("cloud_color_rgb.pcd", *cloud_color_rgb);

// Find PointXYZRGBNormal

  // Normal Estimation

    pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> n;
    pcl::PointCloud<pcl::Normal>::Ptr normals (new pcl::PointCloud<pcl::Normal>);
    pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZRGB>);

    tree->setInputCloud (cloud_color_rgb);
    n.setInputCloud (cloud_color_rgb);
    n.setSearchMethod (tree);
    n.setKSearch (50);
    n.compute (*normals);
```

```
  // Concatenate the XYZRGB and normal fields*
     pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloud_with_normals (new pcl::PointCloud<pcl::PointXYZRGBNormal>);
     pcl::concatenateFields (*cloud_color_rgb, *normals, *cloud_with_normals);

     pcl::io::savePCDFileASCII ("cloud_with_normals.pcd", *cloud_with_normals);


     /*
      pcl::search::KdTree<pcl::PointXYZRGBNormal>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZRGBNormal>);
      tree->setInputCloud (cloud_with_normals);

      std::vector<pcl::PointIndices> cluster_indices;

      pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
      ec.setClusterTolerance (0.02); // 2cm
      ec.setMinClusterSize (100);
      ec.setMaxClusterSize (25000);
      ec.setSearchMethod (tree);
      ec.setInputCloud (cloud_with_normals);
      ec.extract (cluster_indices);
     */
```

```
/*

// Find sphere

  //RANSAC segmentation
  pcl::ModelCoefficients::Ptr coefficients(new pcl::ModelCoefficients);
  pcl::SACSegmentationFromNormals<pcl::PointXYZRGBNormal, pcl::PointXYZRGBNormal> segmentation;

  segmentation.setInputCloud(cloud_with_normals);
  segmentation.setInputNormals(cloud_with_normals);
  segmentation.setModelType(pcl::SACMODEL_NORMAL_SPHERE);
  segmentation.setMethodType(pcl::SAC_RANSAC);
  segmentation.setDistanceThreshold(100);
  segmentation.setOptimizeCoefficients(true);
  segmentation.setRadiusLimits(0.1, 0.15);
  segmentation.setEpsAngle(15 / (180/3.141592654));
  segmentation.setMaxIterations(1000000);

  pcl::PointIndices inlierIndices;
  segmentation.segment(inlierIndices, *coefficients);

if (inlierIndices.indices.size() == 0)
    ROS_INFO("RANSAC nothing found");
  else
  {
    ROS_INFO("RANSAC found shape with [%d] points", (int)inlierIndices.indices.size());
    for (int c=0; c<coefficients->values.size(); ++c)
        ROS_INFO("Coeff %d = [%f]", (int)c+1, (float)coefficients->values[c]);

    // mark the found inliers in green
    for (int m=0; m<inlierIndices.indices.size(); ++m)
    {
        cloud_with_normals->points[inlierIndices.indices[m]].r = 0;
        cloud_with_normals->points[inlierIndices.indices[m]].g = 255;
        cloud_with_normals->points[inlierIndices.indices[m]].b = 0;
    }
  }
```

```
        // Find centroid
            Eigen::Vector4f centroid;
            pcl::compute3DCentroid(*cloud_with_normals, centroid);

            std::cout << "The XYZ coordinates of the centroid are: ("
                      << centroid[0] << ", "
                      << centroid[1] << ", "
                      << centroid[2] << ")." << std::endl;



            // Publish the data
            pub.publish(*cloud_publish);

        }

        int main (int argc, char** argv){
        // Initialize ROS
        ros::init (argc, argv, "sabir");
        ros::NodeHandle nh;

         // Create a ROS subscriber for the input point cloud
        ros::Subscriber sub = nh.subscribe ("input", 1, cloud_cb);

        // Create a ROS publisher for the output point cloud
        pub = nh.advertise<sensor_msgs::PointCloud2> ("output_1", 1);



        // Spin
        ros::spin ();
        }
```

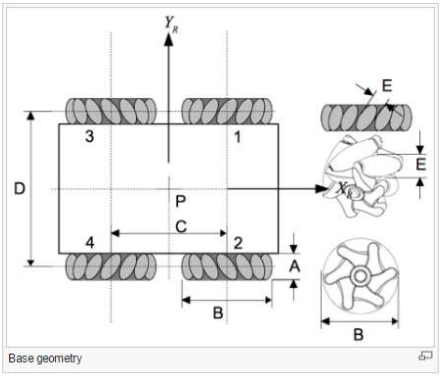
ANNEX B. KUKA YOUBOT DETAILED SPECIFICATIONS

Base

General Info

- Overall Weight : ~20 kg
- Permissible Payload : 20 kg
- Overall Length : 580 mm
- Overall Widht : 380 mm
- Overall Height : 140 mm
- Minimum velocity : 0.01 m/s
- Maximum velocity: 0.8 m/s
- Power supply: 24 V
- Communication: EtherCad

Detailed base geometry



Base geometry

A = 74,87 mm B = 100 mm C = 471 mm D = 300,46 mm E = 28 mm (max diameter of the roll)

- Ground Clearance : 20 mm

Wheels

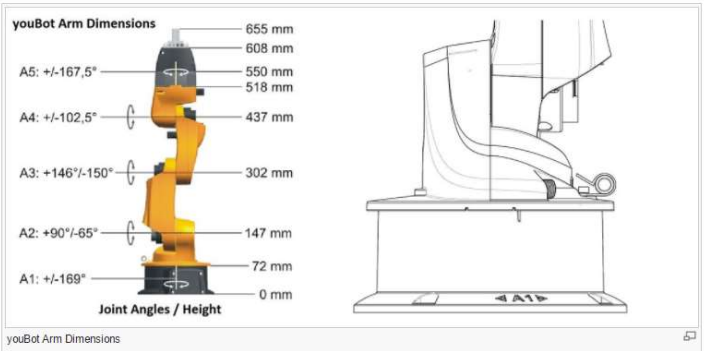
- Type : Omni-directional
- Number of wheels : 4
- Diameter : 47,5 mm

Arm

General Info

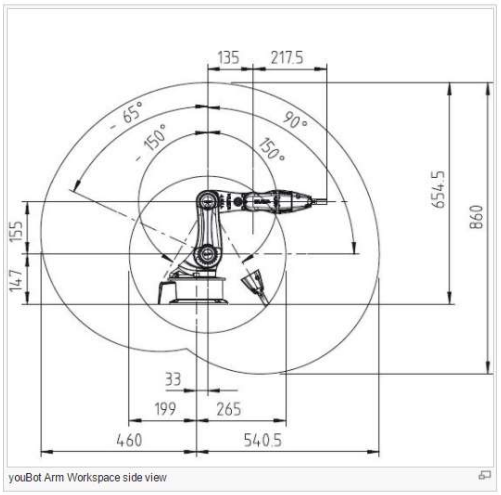
- No. of Axis : 5
- Height : 655 mm
- Work envelope : 0.513 m2
- Weight : 5.3 kg
- Payload : 0.5 kg
- Structure : Magnesium Cast
- Position repeatability : 1 mm
- Communication : EtherCAT
- Voltage connection : 24V DC
- Drive train power limitable to : 80 W
- Axis Speed : 90 deg/s
- Encoders threshold: 1.000 Hz

Arm kinematics

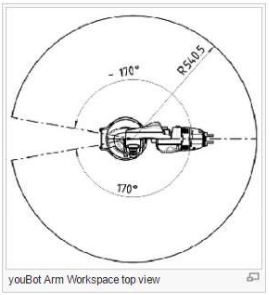


youBot Arm Dimensions

Workspace



youBot Arm Workspace side view



youBot Arm Workspace top view

Arm Actuator

Axis data Range Velocity

```
axis 1   +/- 169°       90 °/s
axis 2   + 90°/- 65°    90 °/s
axis 3   + 146°/ - 151° 90 °/s
axis 4   +/- 102,5°     90 °/s
axis 5   +/- 167,5°     90 °/s
```

Actuator Data

Actuator Data	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Wheel
Motor						
Nominal Voltage (V)	24	24	24	24	24	24
Nominal Current (A)	2.32	2.32	2.32	1.07	0.49	2.32
Nominal Torque (mNm)	82.7	82.7	82.7	58.8		82.7
Terminal Inductance (mH)	0.573	0.573	0.573	2.24	7.73	0.573
Terminal Resistance (Ω)	0.978	0.978	0.978	4.48	13.7	0.978
Moment of Inertia (kg*mm2)	13.5	13.5	13.5	9.25	3.5	13.5
Rated speed (RPM)	5250	5250	5250	2850	2800	5250
Weight (g)	110	110	110	75	46	110
Gearbox						
Reduction Ratio	156	156	100	71	71	26
Moment of Inertia (kg*mm2)	0.409	0.409	0.071	0.07	0.07	0.14
Weight (g)						224
Encoder						
Counts per Revolution	4000	4000	4000	4000	4000	4000

Gripper

- Type : Detachable, 2 fingers
- Gripper stroke : 20 mm
- Gripper range : 70 mm

Battery

Caution: The battery is in general quite sensitive and should never be left connected when the youBot doesn't work because it slowly discharges. It may eventually results in battery failure.

- Type : Maintenance-free lead acid
- Rechargeable : Yes
- Voltage : 24 V
- Capacity : 5 Ah
- Approximate Runtime of youBot : 90 min

Internal PC

- Type : Mini-ITX
- CPU : Intel Atom D510 Dual Core 1.66 GHz
- RAM : 2 GB; SO-DDR2-667 200PIN
- Hard Drive : 32 GB SSD
- Ports : 6 x USB 2.0, 1 x VGA, 2 x LAN (1 available)
- DC Input : 12 V

Motor controller

All used motor controller boards came from TMCM series by Trinamic.

ANNEX C. KINECT DETAILED SPECIFICATIONS

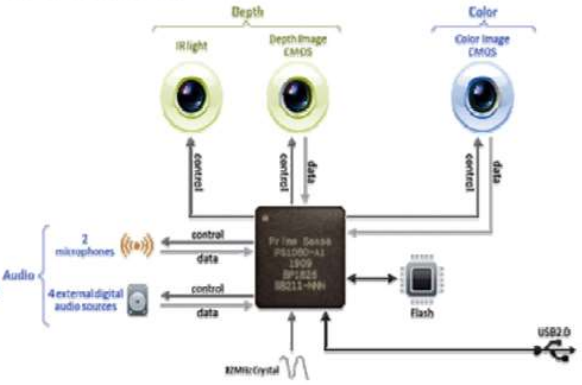
Main components

Video
- Color CMOS camera
- Infrared (IR) CMOS camera
- Infrared projector - 830nm, 60mW laser diode.

Audio
- Four microphones

Tilt control
- Motor
- Accelerometer (3-axes)

Processors & memory
- PrimeSense chip PS1080-A2
- 64 MB DDR2 SDRAM



- Color camera: 640x480 sensor, 640x480@30fps output
- IR camera: 1280x1024 sensor, 640x480@30fps output
- Operation range (depth sensor) = 0.8m - 3.5m
- FOV = 58° H, 45° V, 70° D
- Spatial resolution (@ 2m distance) = 3mm
- Depth resolution (@ 2m distance) = 1cm

Parameter	typical Value
Optical Format	1/6-inch (4:3)
Active Imager Size	2.30mm(H) x 1.73mm(V) 2.88mm Diagonal
Active Pixels	640H x 480V
Pixel Size	3.6µm x 3.6µm
Color Filter Array	RGB Bayer Pattern
Shutter type	Electronic Rolling Shutter (ERS)
Maximum Data Rate/ Master Clock	12 MPS–13.5 MPS/ 24 MHz–27 MHz
Frame Rate (VGA 640H x 480V)	30 fps at 27 MHz
ADC Resolution	10-bit, on-chip
Responsivity	1.0V/lux-sec (550nm)
Dynamic Range	71dB
SNR_{MAX}	44dB

Parameter		Value
Optical format		1/2-inch (5:4)
Active imager size		6.66mm(H) x 5.32mm(V)
Active pixels		1,280H x 1,024V
Pixel size		5.2µm x 5.2µm
Shutter type		Electronic rolling shutter (ERS)
Maximum data rate/ master clock		48 MPS/48 MHz
Frame rate	SXGA (1280 x 1024)	30 fps progressive scan; programmable
ADC resolution		10-bit, on-chip
Responsivity		2.1 V/lux-sec
Dynamic range		68.2dB
SNR_{MAX}		45dB

ANNEX D. PAPER IN CONFERENCE

LITHUANIAN JUNIOR RESEARCHERS: "SCIENCE - FUTURE OF LITHUANIA"

VISION INTEGRATION IN KUKA YOUBOT

Alejandro PEREZ SANMARTIN[1]

[1]*Department of Mechatronics, Faculty of Mechanics, Vilnius Gediminas Technical University, J. Basanavičiaus street 28 - 140, Vilnius, Lithuania*
[1]*alejandro.perez888@gmail.com*

Abstract. This paper presents the integration of Kinect 3D camera in Kuka Youbot and the capabilities that provides to Kuka with a real example case. Specifically, the algorithm that has been developed recognizes a red sphere and provides the coordinates of the sphere to Kuka Youbot Arm. The method has been implemented using Point Cloud Library and color and depth segmentation techniques. Particularly, registered points are processed and transformed to HSV color space avoiding the effect of lightness in color segmentation. Furthermore, sphere modelling is implemented using RANSAC sphere segmentation.

Keywords: Kuka, Youbot, object detection, color segmentation, Kinect, 3D camera. RANSAC, PCL

INTRODUCTION

Nowadays, globalization and the increase of competence has forced companies to innovate and to improve automate processes to reduce direct costs and make them more competitive. Multinationals as Amazon or Toyota has invested big amounts of money in the development of new autonomous robots, specially picking up robots. As a result, Industry 4.0 has exponentially grown up, computerizing the world factory and introducing robots into production and logistics. In addition, the development of new technologies, such as voice recognition, 3D cameras, etc., have made the integration of sensors into robots a very important field of research due to the complexity and the huge challenge that it supposes.

In next future, good qualify workers that can program and control robots will be necessary. This paper has the main objective of the integration of Microsoft Kinect in Kuka Youbot that is available in Vilnius Gediminos Technical University for provide vision to the robot and make a base for future students can improve and practice their robotic skills and get prepare for their future worker life.

Furthermore, it presents the utility of 3D vision with an experimental case: recognition and picking a red sphere using Kuka Youbot and Microsoft Kinect.

Initially, a review of similar methods is done. After that, software and hardware that is used on the project are presented. Then, proposed method is developed and experimental case is explained. Finally, results are analyzed and conclusions are done.

STATE OF ART

Due to the importance and popularity of the use of vision in robotics, some projects and methods have been already developed. In relation to this project, it is possible to classify them in two different groups: control of Kuka Youbot or similar robots and process of images obtained by 3D cameras as Kinect.

In first group, immersive ROS framework in robots [1] or the work in a model for Kuka Youbot[2] are examples of the importance of Kuka Youbot in research. Furthermore, some studies have been done related to inverse kinematics. Inverse kinematics can be solved by numerical or analytical approaches [3], [4], [5]. Numerical methods are mainly used for robotic systems and analytical closed form IK solvers dominate in deliberative planning like trajectory generation and path planning [4], [6].

In second group, sensing systems allow to propose new attractive solutions for robot navigation problems like 3D mapping and localization [7], object recognition [8], 3D modeling [9], visual odometry [10] among others perception tasks. Specifically, there are some projects already developed for detecting objects and applying color segmentation[11], but this project propose a different innovative method incorporating the best method in every step of the algorithm.

SOFTWARE AND HARDWARE FEATURES

The proposal algorithm has been programmed with C++ and use Ubuntu as system operator. As well, Robot Operating System (Hydro version) is used for interact with Kuka Youbot Arm. ROS is a collection of software frameworks for robot software development, for control and communication with the robot.

The sensor device chosen is Microsoft Kinect. Furthermore, some libraries as Point Cloud Library and OpenCV are used for manipulate the data.

It is really important to know the characteristics and limitations of the hardware that is used. YouBot arm is a 5 degree-of-freedom arm. It is a 5 link serial kinematic chain with all revolute joints. The arm is similar to KUKA's larger industrial robot arms. The dimensions of each link of the arm, as well as the range of each joint are shown in Figure 1. Scope of movement of Kuka Youbot Arm is shown in Figure 2.

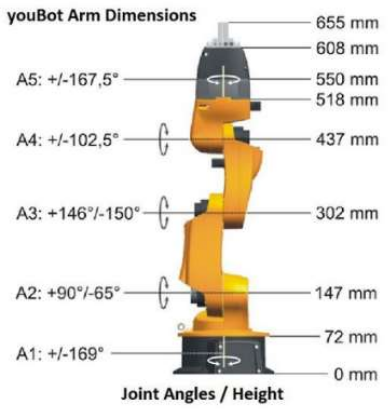


Figure 1: Kuka Youbot Arm features

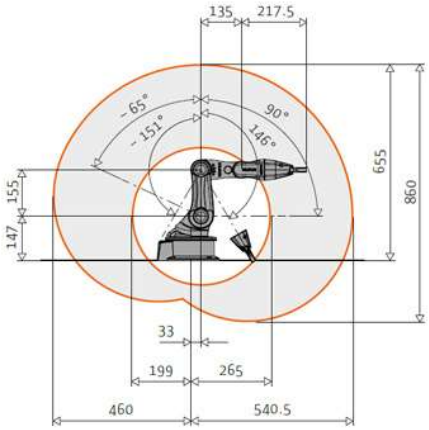


Figure 2: Scope of movement Kuka Youbot Arm

The Microsoft Kinect is a consumer device originally designed by Microsoft as a peripheral for the Xbox game system and Windows PCs. The quality of the depth information produced by the Kinect make it a very attractive sensor for robotics research. As a result, a variety of open-source drivers have been developed for the Kinect which allow one to process the information produced by a Kinect as a cloud of points located in 3D space. Kinect is composed by several sensors shown in Figure 3. Angle and range of vision of Kinect are shown in Figure 4 and 5.

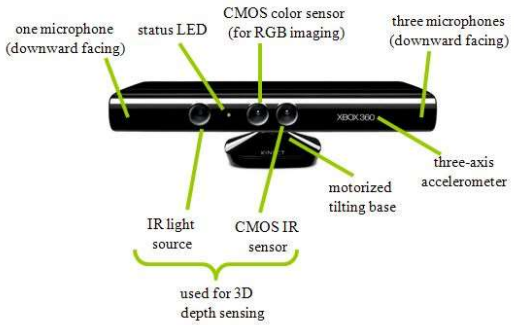


Figure 3: Kinect sensor

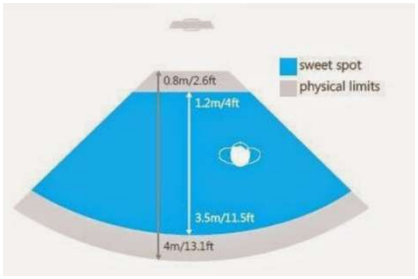


Figure 4: Range of vision Kinect

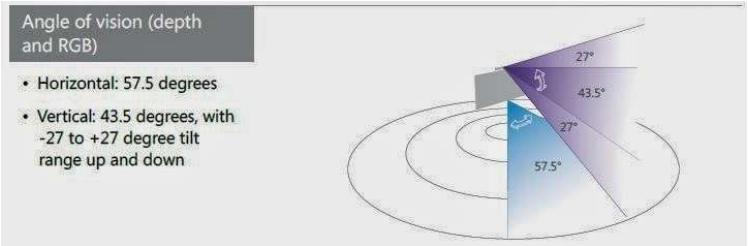


Figure 5: Angle of vision Kinect

PROPOSED METHOD

The block diagram of the proposed method is presented in Figure 6.

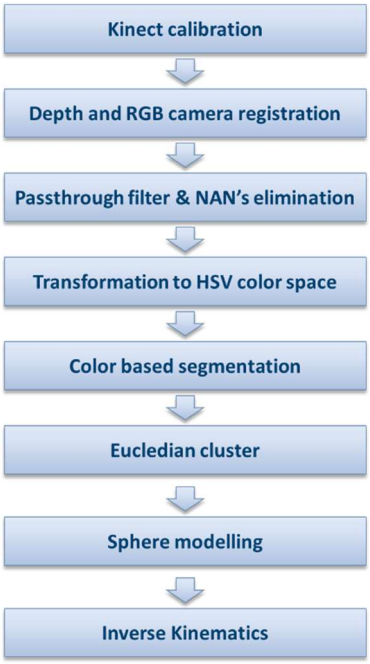


Figure 6: Proposed method

Initially Kinect calibration is done. After that, registration process starts for unify depth and color information in just one point cloud. That process is important because depth sensor and camera RGB are bringing information separately.

Then, Pass-through filter is applied for fix area of action of the Kinect in x, y and z coordinates. After that, Nan's elimination is done. Those two processes will decrease the number of points and make the algorithm faster. Then, RGB information is unpacked using bit shifting methods. Next step is to transform it to HSV space. In Figure 7 are shown characteristics of HSV space.

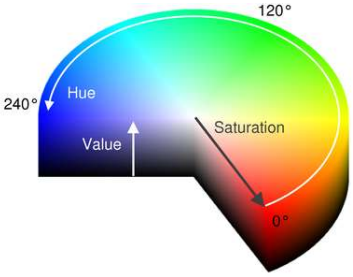


Figure 7: HVS color space

HSV color space is a better representation to classify object color because it separates hue and saturation from value (lightness). Hue and saturation are often constant but lightness will change depending on the lighting strength in the room. So the detection criteria should be discriminative in the hue and saturation but invariant to lightness.

Then color segmentation is implemented. Red values in HSV color are selected as limits, and points with different values are eliminated. Then, Normal Estimation, Eucledian cluster and sphere fitting model are done using RANSAC model sphere segmentation and Point Cloud Library. After that, centroid of the sphere is found and exactly coordinates are sent to the robot.

Finally, inverse kinematics already developed in old projects and shared by Kuka are applied for move the Youbot Arm to the desired position and pick up the sphere.

EXPERIMENTAL CASE AND RESULTS

Experimental case is done for evaluate the accuracy of the method. Red sphere is placed in dark background available for Kinect view and Youbot Arm. The results show that algorithm is working for the color segmentation and detect the red sphere. Main problem appears with fitting the sphere in the model and finding the centroid. As camera can only see one part of the sphere, centroid is approximately found, but not perfectly. That can make the robot fail in picking process.

In future work, it should be added a second camera from other perspective for find the exactly point.

CONCLUSIONS

Importance of robots in nowadays industry makes this project and other researches in mechatronics and robotics important. In this paper, Kinect 3D camera is integrated in Kuka Youbot and the development of an algorithm to show its capabilities is done. Results have shown that Kinect can provide vision to Kuka for recognition and detection of objects. Possible improvements for future work is to integrate a second camera for have a completely and more precise vision of the environment.

REFERENCES

[1] L. Peppoloni, F. Brizzi, C. Avizzano, E. Ruffaldis *Immersive, ROS-integrated Framework for Robot Teleoperation,* PERCRO Laboratory, Tecip Institute Scuola Superiore Sant'Anna

[2] R. Dwiputra, A. Zakharov, R. Chakirov, E. Prassler Modelica Model for the youBot Manipulator Bonn-Rhein-Sieg University of Applied Sciences, Department of Computer Science Grantham-Allee 20, 53757 Sankt Augustin

[3] Khatib, O.; Siciliano, B., editors: Springer Handbook of Robotics, Springer, Berlin, Heidelberg, Germany, 2008.

[4] Craig, J.: Introduction to Robotics Mechanics and Control, Third edition, Chapter 3, Manipulator Kinematics, Pearson Prentice Hall, New Jersey, USA, 2005.

[5] Baker, D.; Wampler, C.: On the Inverse Kinematics of Redundant Manipulators, International Journal of Robotics Research, vol.7, No.2, 1988

[6] Berenson, D.; Srinivasa, S.; Ferguson, D.; Romea, A.; Kuffner, J.: Manipulation planning with workspace goal regions, Proc. of the IEEE International Conference of Robotics and Automation (ICRA-2009), IEEE Press, 1999.

[7] P. Henry, M. Krainin, E. Herbst, X. Ren, D. Fox, Rgbd mapping: Using depth cameras for dense 3d modeling of indoor environments, in: In RGB-D: Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS, 2010

[8] K. Lai, L. Bo, X. Ren, D. Fox, Sparse distance learning for object recognition combining RGB and depth information, in: 2011 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2011, pp. 4007–4013.doi:10.1109/ICRA.2011.5980377.

[9] P. Henry, M. Krainin, E. Herbst, X. Ren, D. Fox, Rgbd mapping: Using depth cameras for dense 3d modeling of indoor environments, in: In RGB-D: Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS, 2010. Y. Ohta, T. Kanade, T. Sakai, Color information for region segmentation, Computer Graphics and Image Processing 13 (1) (1980) 222 – 241.

[10] F. Steinbruecker, J. Sturm, D. Cremers, Real-time visual odometry from dense rgb-d images, in: Workshop on Live Dense.

Reconstruction with Moving Cameras at the Intl. Conf. on Computer Vision (ICCV), 2011.